Solving Empty Result Problem in Keyword Search over Relational Databases

Masters Thesis

Hu Junfeng

hujunfeng@comp.nus.edu.sg

supervised by

Associate Professor Chan Chee Yong

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

April 2012

Abstract

Keyword search over relational databases provides a simple and intuitive query interface for normal users to retrieve information from databases. Most of the existing keyword search systems for relational databases use foreign key references to connect tuples. In this case, an answer to a keyword query is typically a joining network of tuples that are connected via a series of foreign key references, and contain all the keywords in the query. However, if no such joining network of tuples exists, which means foreign key relationship fails to connect tuples to cover all the keywords, no result would be returned to users. This problem is called the empty result problem, which usually would disappoint the user. Instead of returning nothing, in this thesis, we propose a solution to automatically find approximate answers that contain all the keywords in the query.

Acknowledgement

I would like to take this opportunity to show my deepest gratitude to people who help and support me. Without them, I would not have been able to finish my thesis.

First of all, I would like to thank my supervisor Prof. Chan Chee Yong for his continuous support, guidance and patience with me. During my work with him, Prof. Chan showed his highly professionalism and very strong knowledge. I have learned a lot from him.

I would also like to thank my parents who love and support me in all kinds of ways even when I could not be around them for most of the time.

Last but not the least, I would like to thank all the superiors involved in the evaluation of this thesis. I appreciate them taking their precious time to evaluate my work. I take all responsibility for any errors or inadequacies that may remain in this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Keyword Search (KWS) is a simple and flexible query interface. The popularization of web search engines made keyword search become one of the most acceptable search manner for normal users. Naturally, people want to bring keyword search interface into relational databases, which has conventionally use Structured Query Language (SQL). From end users' perspective, *Keyword Search over Relational Databases* (KWS-R) has several benefits over traditional SQL queries. First, users do not need to know the underlying schema of the database. Second, users do not need to learn the complex SQL syntax and create numerous complicated SQL statements for a simple query. Powered with keyword search interface, relational databases would be much more useful to new users, especially for those without any database background.

The current approach to KWS-R is modeling the database as a directed data graph, in which nodes are tuples and edges are foreign key references between tuples. An answer to a keyword query is a subgraph of the data graph that contains all the keywords in the query. If the data graph is connected, which means every pair of distinct tuples in the graph can be connected by a sequence of foreign key references, the user will be guaranteed to have an answer if all keywords are contained in some tuples. However, this is not always true, especially when modeling a complex database in real world. Instead of a single

connected graph, it is more likely that a number of distinct graphs will be constructed. For example, we use two datasets in our experiments: IMDb (516MB, 6 relations, 1,673,074 tuples) and Wikipedia (550MB, 6 relations, 206,318 tuples). Both datasets are from a framework proposed by Coffman et al. [6] for evaluating keyword search in relational databases. The data graph of the IMDb dataset consists of 317 disconnected graphs, while the data graph of Wikipedia also has 19 disconnected graphs. Therefore, it is likely that given a query, tuples that contains keywords are distributed across many graphs, and no single graph contains all keywords.

Even if we can find a graph that contains all keywords, there is also another problem. For the data graph of the IMDb dataset, the average length of the shortest path between any two tuples is 21. Given two tuples, the distance between them could be so long that it is meaningless to connect all these tuples into results, and it is also very costly to do so in existing KWS-R systems. There is usually a limit of the size of results. A typical size limit would be less than 10 tuples. Therefore, even in a graph that contains all keywords, there possibly does not exist a subgraph of tuples that contain all keywords where its size is less than the limit.

In both cases, no result would be returned to the user. This problem is called the *empty result problem*. Empty results usually disappoint the user and reduce the usability of the system, because it does not provide any useful information. The cause of the empty result problem is that there is only one type of connection, i.e., foreign key reference, to represent the relationship between tuples in databases.

Therefore, the essential idea of our solution to the empty result problem is add a new type of relationship that connects tuples from disconnected graphs. We use the *tuple similarity* as the new relationship to connect tuples. In other words, two tuples are connected if they are similar. With the new relationship, we present an algorithm that find a connection between partial answers, each of which only contains a portion of keywords in the query, but they together

contain all keywords. Then connected partial answers are combined to form final answers.

In the rest of the thesis, we first discuss related work in Section 2. In Section 3, we formally define the empty result problem and present our solution. We show our experiment to evaluate the effectiveness of our solution in Section 4 and conclude the thesis in Section 5.

Chapter 2

Related Work

The background of our work is keyword search over relational databases. In the following sections, we will go through the fundamentals of KWS-R. Among the existing systems of KWS-R, the architectures typically can be divided into two main components:

1. An **Engine** that generates candidate answers.
2. A **Ranker** that ranks candidate answers by evaluating the scoring function.

We discuss the basic concepts in Section 2.1, and the engines of KWS-R in existing solutions in Section 2.2 and 2.3, and finally the rankers of KWS-R in Section 2.4. Another related work is the general empty result problem in relational databases. Existing works towards this problem are discussed in Section 2.5. We also discuss the related work about tuple similarity in Section 2.6.

2.1 Basic Concepts

Keyword search was first popularized in text search. It is a very different context from relational databases. In text search, the data is a set of documents/web

pages with little structure; while in KWS-R, the data is a list of tables, where each table consists of tuples, and each tuple consists of attributes. In other words, they are highly structured. In text search, an answer to a query is simply a document, while in KWS-R, an answer is more complex. In the literature, the most widely used answer model is a list of joined tuples, proposed in [1, 17]. We formally define the data model, the answer model and the query model in KWS-R as follows.

Data Model

A relational database R is considered as a directed graph $G(V, E)$. Each tuple t_i in R corresponds to a node t_i in V; each foreign key reference from t_i to t_j corresponds to an edge $t_i \rightarrow t_j$ in E. Without any ambiguity, in the rest of the thesis, we use tuple and node interchangeably to refer a tuple in R or a node in G. The schema of R is also considered as a directed graph $G_S(V_S, E_S)$. Each relation is a vertex and each primary-foreign key reference between two tables represents an edge. G is called the *data graph* while G_S is called the *schema graph*.

Query Model

A m-keyword query Q is a set of keywords of size m, $\{k_1, k_2, \cdots, k_m\}$. A node in G that contains at least one of the keywords in Q is called a *keyword node*, or *keyword tuple*. A node that do not contain any keywords in Q is called a *free node*, or *free tuple*.

Answer Model

A list of joined tuples is defined as a *Joining Network of Tuples* (JNT) in [17].

Definition 1 (Joining Network of Tuples)**.** *A joining network of tuples in a data graph is a graph of tuples where each edge presents a foreign key reference between the tuples at two ends of the edge.*

In the data graph G for a relational database R, a JNT is actually a

subgraph of G. For a query Q, the result of Q is a set of all possible JNTs that contain all the keywords in Q. According to different semantics, more constraints will be applied on JNTs for them to be in the final results. We will discuss these constraints very soon.

2.2 KWS-R Engine

In the last decade, KWS-R has been intensively studied in the database community. A lot of algorithms and systems are proposed, however almost all of them fall into the following two categories:

1. **Graph-Based KWS-R** To materialize the data graph G in memory and find the satisfied JNTs using graph algorithms;

2. **Schema-Based KWS-R** To use the schema graph G_S to find all the subgraphs of G_S, that would possibly generate satisfied JNTs in G, then convert these candidate subgraphs into SQL queries, and finally evaluate SQL queries on the underlying DBMS.

Among existing systems, BANKS [4], BANKS2 [20], BLINKS [14], Min-Cost [9], Golenberg [11], Kimelfeld [22], ObjectRank [3, 16], EASE [24], Progressive [23, 25], and Community [35] are Graph-Based. DBXplorer [1], DISCOVER [17], DISCOVER2 [15], Sayyadian [37], Liu [26], SPARK [27], and Xu [39] are Schema-Based. S-KWS [29, 30] and PowerDB [34] use both approaches in their systems.

Besides two different ways of designing algorithms, the system evaluation is also emphasized differently among these works. Many works [9, 11, 22, 23, 25, 35] mainly focus on the efficiency of their systems, while some [26, 39] works only improve the effectiveness of KWS-R. There are also many works [4, 20, 14, 24, 3, 27, 15] taking both efficiency and effectiveness into consideration. Figure 2.1 shows the comparison of these systems in the two dimensions. Note that this figure compares the main focus of their work on KWS-R, but not compare their

algorithms. An interesting observation is that most Graph-Based systems focus on efficiency and no work is dedicated to effectiveness; however, there are two dedicated works in effectiveness in Schema-Based KWS-R, though efficiency is still the main consideration of most Schema-Based systems.

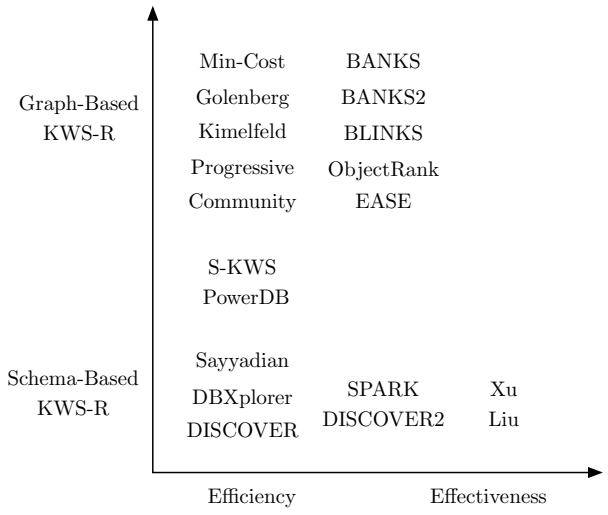


Figure 2.1: Comparison of Different Systems of KWS-R

2.2.1 Graph-Based KWS-R

Graph-Based keyword search over relational databases is introduced in BANKS [4], and later many works [35, 34, 24, 16, 20, 14, 3, 9, 11, 22, 23, 25, 29, 30, 8] adopt this approach. These Graph-Based KWS-R systems assume that the data graph G is materialized in the memory.

There are at least four semantics of JNTs proposed based on Graph-Based KWS-R: *Steiner tree semantics* [4, 20], *distinct root semantics* [14], *distinct core semantics* [35], and *r-radius Steiner graph semantics* [24].

Steiner Tree Semantics

Given a m-keyword query Q, for each keyword k_i in Q, we can find all the keyword nodes in the data graph G that contain k_i. We denote such set

of keyword nodes as K_i, for each k_i. In this semantics, an answer JNT is a minimal rooted directed tree containing at least one keyword node from each K_i. It may also contains free nodes, and is therefore a Steiner tree [18].

The problem with the Steiner tree semantics is that it may result in duplicate JNTs. Consider an answer JNT $t_1^{k_1} \leftarrow t_4 \leftarrow t_2^{k_2} \rightarrow t_5 \rightarrow t_3^{k_3}$ as an example, where $t_1^{k_1}, t_2^{k_2}, t_3^{k_3}$ are keyword nodes containing k_1, k_2, k_3 respectively, and t_4, t_5 are free nodes. Obviously, here $t_2^{k_2}$ is the root. Since every two tuples that have a forward edge also have a backward edge and a rooted directed tree has only one root, we actually have four more answer JNTs for these five tuples: $t_1^{k_1} \rightarrow t_4 \rightarrow t_2^{k_2} \rightarrow t_5 \rightarrow t_3^{k_3}$, $t_1^{k_1} \leftarrow t_4 \leftarrow t_2^{k_2} \leftarrow t_5 \leftarrow t_3^{k_3}$, $t_1^{k_1} \leftarrow t_4 \rightarrow t_2^{k_2} \rightarrow t_5 \rightarrow t_3^{k_3}$, and $t_1^{k_1} \leftarrow t_4 \leftarrow t_2^{k_2} \leftarrow t_5 \rightarrow t_3^{k_3}$, where the roots are $t_1^{k_1}$, $t_3^{k_3}$, t_4, and t_5 respectively. However all the five JNTs are the same from users' perspective. In general, for every answer JNT T with $|T|$ nodes, there are $|T|$ duplicates of T, each of which have one of the nodes in T as the root. Therefore, in the Steiner tree semantics, there can be as many as $O(2^{|E|})$ JNTs theoretically, where $|E|$ is the number of edges in G.

Distinct Root Semantics

BLINKS [14] proposes the distinct root semantics that does not suffer from the issue of the Steiner tree semantics. It overcomes the duplicate problem by defining a distinct root for each JNT. Specifically, an answer JNT, assuming that t_r is the root, satisfies the following conditions: (1) the answer contains at least one keyword node from each K_i; (2) among all the keyword nodes $t_j^{k_i}$ for each k_i, one of $t_j^{k_i}$ is chosen such that the distance from the root t_r to this $t_j^{k_i}$ is minimum; (3) the minimum distance is less or equal to a user given parameter `Dmax`. Intuitively, the further a keyword node is away from the root node, the less interesting it is, so the last condition is used to limit the number of answers.

The benefit of this semantics is that for each root node, there is only one JNT that satisfies the condition. Consider the above example and suppose t_1 is the root. If the nearest keyword node containing k_3 is another node t_6, for example in a path $t_6^{k_3} \leftarrow t_1^{k_1}$, then the nearest keyword node containing k_2 is still $t_2^{k_2}$. In that case, the JNT $t_1^{k_1} \rightarrow t_4 \rightarrow t_2^{k_2} \rightarrow t_5 \rightarrow t_3^{k_3}$ will not contribute to the results, but $t_6^{k_3} \leftarrow t_1^{k_1} \rightarrow t_4 \rightarrow t_2^{k_2}$ does. In theory, there are at most $|V|$ number of JNTs in the final results, where $|V|$ is number of nodes in G.

Obviously, the distinct root semantics produces a subset of results that the Steiner tree semantics does. The former reduces the number of results by limiting the results to one answer for each node that is deemed the root node. It removes many duplicate answers and also avoid the "hub" problem, where one node connects to many keyword nodes and become the root of a huge number of answers. Users may not be interested in all the answers from the "hub", but only the most concise and informative one as the distinct root semantics gives.

Distinct Core Semantics

However, people also argue that what if users want more information than that a single rooted tree can provide. Hence, more complex semantics, like Distinct Core Semantics, are proposed to show users more information. In Distinct Core Semantics, an answer JNT is a multi-center subgraph of G, called a *community* [35], which is defined on a set of keyword nodes. Specifically, the set of nodes V in a community is a union of three subsets, $V_k \cup V_c \cup V_p$, where V_k is a set of keyword nodes that contain at least one node from each K_i, V_c is a set of center nodes that for each center node $t_c \in V_c$, the distance from t_c to every $t_k \in V_k$ is less or equal to a user given parameter `Dmax`, and V_p is a set of nodes that appear on the shortest paths from each $t_c \in V_c$ to each $t_k \in V_k$. The set of edges in a community consist of all the edges that appear on the shortest paths from

each $t_c \in V_c$ to each $t_k \in V_k$.

Given a community, the set of keyword nodes V_k is called the *core*, as V_k uniquely determines the community by definition. In other words, there are no two communities that share the same core.

In contrast to the distinct root semantics, this semantics does not add any new answers in terms of combinations of keyword nodes. However, a community does provide new information by adding all the center nodes with regard to the same set of keyword nodes, i.e., the core. For each possible set of keyword nodes V_k, the distinct root semantics gives a single root/center that connect all of them, while the community of V_k actually combines all the single rooted trees of the same V_k into one subgraph. [35] believes that such combination presents more interesting information.

r-radius Steiner Graph Semantics

Inspired from the Steiner tree problem, EASE [24] introduced the r-radius Steiner graph to model keyword search problem over relational databases as well as semi-structured data (e.g., XML) and unstructured data (e.g., text documents).

In this semantics, an answer JNT becomes a r-radius Steiner graph, which is defined as follows. First, the *centric distance* between a node t in G and a subgraph G' of G is the maximum value among the shortest distances from t to any node in G'. Second, the *radius* of a subgraph G' is the minimum value among the centric distances between any node in G to G'. Third, a r-radius graph has exactly the radius of r. Finally, for a keyword query Q, the r-radius Steiner graph is the r-radius graph that contains at least one keyword node from K_i. (EASE do not require that it must contain all the keywords in Q, however for consistency with our discussion, we only consider the situation that the results contain every keyword, i.e., AND semantics of keyword search.) As in a Steiner tree, it may also contain free nodes.

The r-radius Steiner graph semantics also does not add any new answers in terms of keyword nodes. However, as the distinct core semantics, it adds more nodes related to a set of keyword nodes. In contrast to the distinct core semantics, which uses the center nodes to extend its answers, this semantics includes all the nodes that fall in the same r-radius graph as the set of keyword nodes do. This is like a new relationship that tuples are connected. Therefore, for a given set of keyword nodes, additional tuples followed by the relationship may provide users more information.

For each semantics, a particular algorithm is designed to search for results. In the following, we describe algorithms under the Steiner tree semantics and the distinct root semantics, since these are the most classical algorithms in Graph-Based KWS-R.

BANKS search for JNTs (Steiner tree semantics) using the *backwards search algorithm*. BANKS first find all the keyword nodes in the data graph G, using a inverted index. Recall that K_i denotes the set of keyword nodes containing keyword k_i in Q. Let K be the set of all K_i. The backwards search algorithm runs $|K|$ copies of Dijkstra's single source shortest path algorithm concurrently, with each keyword node in K_i being the source. Every time a node is visited, it records the source node and the keywords of the source. Once a node v has been visited by all the keywords, a new result is found by constructing a rooted directed tree with v as the root and all the reverse paths to the sources. This is a simple description of the algorithm. The problem to find the optimal Steiner tree is known to be NP-hard. Therefore, the top result of BANKS is only an approximation of the optimal Steiner tree. The results found using the backwards search algorithm is also not complete, as it only considers the shortest path from the root of a tree to nodes containing keywords.

The backwards search algorithm would potentially visit an unnecessarily large number of nodes if (1) the query contains some frequently occurring keyword (e.g., *database* in DBLP); (2) some node has a large number of incoming edges. BANKS2 [20] overcome this issue by introducing *bidirectional search*

algorithm that search the data graph both *backwards*, as in backwards search algorithm, from keyword nodes, and *forwards* from potential roots that are visited by backwards searching algorithms. BLINKS [14] further accelerates the search processing by reducing the search space through a bi-level index of the data graph. A bi-level index is a precomputed two-level index built by first partitioning graph, and then building indexes inside partitions as well as an index of partitions.

2.2.2 Schema-Based KWS-R

Given that relational databases are modeled as graphs, it is natural to solve KWS-R using graph algorithms. However, by that way, we abandon all the functionalities provided by today's sophisticated database management systems. Therefore, the Schema-Based approach, on the other hand, utilizes the schemas of relational databases to translate a keyword query into a series of SQL statements, afterwards which are executed directly on the DBMS to retrieve the results. DISCOVER [17] and DBXplorer [1] are the first two systems that use the Schema-Based approach, which is later adopted by many works [15, 26, 27, 29, 37, 39]. Compared to DISCOVER, DBXplorer is much simpler, because it only allows exact match between a keyword and an attribute value, and it also does not consider cases in which two tuples are from the same relation. Therefore, we mainly discuss DISCOVER in the rest of the section.

In DISCOVER, a relational database is still modeled as a data graph G as Graph-Based KWS-R does, but the data graph G is never materialized and only remains as conceptual. However, the schema graph G_S, which is never used in Graph-Based KWS-R, is materialized to generate Candidate Networks, which are to be defined very soon.

There are only one semantics of JNT used as answers in Schema-Based KWS-R. It is called *Minimal Total Joining Network of Tuples* (MTJNT), which is defined as follows.

Definition 2 (Minimal Total Joining Network of Tuples)**.** *A minimal total JNT*

is a JNT that satisfies the following two conditions: (1) it is total, which means it contains all the keywords in the query Q, and (2) it is minimal, which means it will not be total if any node is removed.

In fact, MTJNT is the same as the Steiner tree semantics in Graph-Based KWS-R. A node in MTJNT with degree of 1 is called a *terminal* node. By definition, MTJNT has a nice property that each terminal node contains at least one distinct keyword, because if a terminal node only contains keywords that other nodes have, it can be removed, which contradicts the definition of MTJNT. This property is very helpful in pruning Candidate Networks.

The general idea of finding MTJNTs in tables is that firstly it uses the schema graph G_S to find all the joining networks of relations that possibly generate MTJNTs; secondly, it evaluates these candidate joining networks of relations to retrieve all the satisfied MTJNTs. G_S is a graph of relations $R_S = \{R_1, R_2, \dots\}$ of the relational database R. For each relation $R_i \in R_S$, R_i will contribute a tuple to an answer MTJNT if and only if R_i contains some keywords in the query. We define *Expanded Schema* as follows.

Definition 3 (Expanded Schema)**.** *For a relation S and a set of keyword $K \subseteq Q$, the expanded schema $S\{K\}$ is a subset of S containing tuples that contain exactly all the keywords in K, no more, no less.*

For example, assuming $Q = \{k_1, k_2, k_3\}$, $S\{k_1, k_3\}$ is the set of tuples that contain only k_1, k_3 but no k_2. For convenience, $S\{\}$ is the set of all *free tuples* in S. Thus, for each relation $R_i \in R_S$, there are in total $2^{|Q|}$ expanded schemas, where $|Q|$ is the number of keywords in the query. In the above example, $Q = \{k_1, k_2, k_3\}$, there are eight expanded schemas for S, that is, $S\{\}$, $S\{k_1\}$, $S\{k_2\}$, $S\{k_3\}$, $S\{k_1, k_2\}$, $S\{k_2, k_3\}$, $S\{k_1, k_3\}$, $S\{k_1, k_2, k_3\}$.

We then define *Expanded Schemas Graph* and *Candidate Network* (CN) as follows.

Definition 4 (Expanded Schemas Graph)**.** *For a schema graph G_S and a query Q, an* expanded schemas graph $\mathbf{G_{ES}(Q)}$ *is constructed by (1) nodes are the*

expanded schemas that are not empty; (2) every two nodes have an edge iff their corresponding relations have an edge in G_S.

Definition 5 (Candidate Network)**.** *A candidate network is a subgraph of expanded schemas graph $G_{ES}(Q)$ that satisfies two conditions: (1) it is total, which means it contains all the keywords in Q; (2) it is minimal, which means it will be not total if any node is removed.*

Intuitively, a candidate network is a joining network of relations that possibly generate MTJNTs. Observe that CN has a similar definition as MTJNT. Actually, CN can be considered as the projection of MTJNT onto the expanded schema.

With these definitions, we move on to discuss the query processing of Schema-Based KWS-R. There two main steps:

1. **Candidate Networks Generation**. In this step, first, an expanded schema graph $G_{ES}(Q)$ is built from the schema graph G_S and the query Q. From $G_{ES}(Q)$, a set of candidate networks is generated, which is required to be *complete* and *duplicate-free*.

2. **Candidate Networks Evaluation**. With input of CNs, this step creates a SQL execution plan, which is then executed on the DBMS to obtain the results.

The key challenge of the candidate networks generation is that the set of CNs must be *complete* and *duplicate-free*. By complete, the set of CNs generates all the MTJNTs. By duplicate-free, every two CNs are not isomorphic to each other. DISCOVER produces a complete and duplicate-free set of CNs by enumerating all subgraphs of $G_{ES}(Q)$ that does not violate any pruning rules. Three pruning rules used in DISCOVER are listed as follows.

1. Prune duplicate CNs.

2. Prune CNs that are not minimal, i.e., CNs having a leaf node of form $R_i\{\}$, which does not contain any keywords.

3. Prune CNs that are of form $R_i\{K_1\} \leftarrow R_j\{K_2\} \rightarrow R_i\{K_3\}$, where $K_1, K_2, K_3 \subseteq Q$ and $K_1 \neq K_2 \neq K_3$. Note that in such form, a primary key is defined in R_i and a foreign key is defined in R_j pointing to the primary key. Any tuple in $R_j\{K_2\}$ that has a foreign key referring to a tuple in R_i must point to the same tuple in R_i. However, the same tuple cannot appear in two different sets, as $R_i\{K_1\} \cap R_i\{K_3\} = \emptyset$. As a result, CNs of this form cannot generate valid MTJNTs.

DISCOVER's algorithm [17] enumerates all subgraphs in a breath-first traversal of $G_{ES}(Q)$. Specifically, the algorithm first randomly pick a keyword $k \in Q$, and start traversals from all the nodes (expanded schemas) in $Q_{ES}(Q)$ that contains k. In each round, a subgraph is pruned if it satisfies the pruning conditions, otherwise it is outputted as a new CN if it contains all keywords, otherwise it is expanded for by adding an adjacent node in $G_{ES}(Q)$ to a node of it. There is also a parameter `Tmax` to limit the size of CNs, because it is not meaningful if CN is too large in size.

Actually, in DISCOVER, duplicate CNs are removed through a post-processing step, but cannot be avoided in the breadth-first traversal algorithm. Markowetz et al. [29] propose an improved algorithm that guarantees to generate duplicate-free CNs directly. The basic idea is to enumerate subgraphs in a unique pre-order traversal.

The second phase is to evaluate the candidate networks, i.e., converting candidate networks into query trees and creating a SQL execution plan. A naive plan is to simply create SQL queries for each candidate network and run the queries independently. This method has a big performance problem, because candidate networks typically share same join subexpressions, and thus same join operations might be run many times. Therefore, an efficient execution plan should store the common join expressions as intermediate results and reuse them whenever possible. Unfortunately, in DISCOVER [17], the problem of finding the optimal execution plan is proved to be NP-complete. DISCOVER gives a greedy algorithm that produces a near-optimal execution plan, using two

heuristics: (1) subexpressions shared by most CNs should be evaluated first; (2) subexpressions that generate small number of results should be evaluated first.

2.3 KWS-R Ranker

Ranking results is one of the major challenges in keyword search over relational databases, because the number of results usually will be very large, and the user is only interested in a small number of the most relevant results. In the literature, people have proposed several ways to rank the results. In this section, we show these ranking methods in KWS-R.

Although JNT is used as an answer to a keyword query in both Graph-Based KWS-R and Schema-Based KWS-R, the ranking methods of them are different. Graph-Based KWS-R mostly uses a ranking method that are based on graph, such as PageRank, because with the whole data graph in memory, it is easy to consider the weight of a node or an edge globally. For example, computing the number of incoming edges of a node is easy in Graph-Based KWS-R, however it is difficult in Schema-Based KWS-R. On the other hand, Schema-Based KWS-R mostly uses IR-style ranking method, that is to consider the actual values of tuples in the databases. IR-style method can also be used in Graph-Based KWS-R.

2.3.1 Ranking Methods in Graph-Based KWS-R

BANKS [4] consider that tuples and edges in an answer JNT are usually not the same importance, therefore they assign weights to each node (PageRank-style, for example, the PageRank of a node is defined recursively and depends on the number and PageRank metric of all incoming nodes. A note that is connected to by many nodes with high PageRank receives a high rank itself.) and each edge (based on how related the two tuples are), and then combine them to compute a final score for ranking.

In BANKS, the overall score of an answer is defined by three parts: (1) an

expression of the overall score of all the nodes in the answer JNT; (2) an expression of the overall score of all the edges in the answer JNT; (3) combination of (1) and (2) to get the overall score of the answer JNT.

For node weights, in BANKS, each node u in the graph is assigned a weight $N(u)$ reflecting the prestige of the node. Specifically, $N(u)$ is defined to be a PageRank-style function of the indegree of u. The overall score of nodes is

$$Nscore = \sum \log(1 + N(u)/Nmax),$$

for each u in the answer JNT, where N_{max} is the maximum node weight in the graph.

For edge weights, a forward edge $u \rightarrow v$ is assigned a weight w_{uv} based on the strength of the foreign key reference between the two relations. BANKS assumes the strengths of foreign key references are manually judged by domain experts or database administrators. For example, the link between `Papers` and `Writes` would have stronger connection than the link between `Papers` and `Cites`. For a backward edge $u \leftarrow\!\!- v$, the weight is $w_{vu} = w_{uv} \log_2(1 + \text{indegree}(v))$. The overall score of edges is

$$Escore = \frac{1}{1 + \sum \log(1 + w_u v / w_{min})},$$

for each forward and backward edge in an answer JNT, where w_{min} is the minimum edge weight in the graph.

For the final part, BANKS presents several ways for combination, either by addition or by multiplication. Let T be an answer JNT and Q a query, the scoring functions used in BANKS are:

$$Score(T, Q) = (1 - \lambda)Escore + \lambda Nscore \quad \text{or}$$

$$Score(T, Q) = Escore * Nscore^{\lambda}$$

where λ is a factor to control their relative weightage.

ObjectRank [3] proposed another more complicated node weights. Instead of only considering the global and static importance of nodes (as PageRank), ObjectRank additionally considers the relevance of nodes to the keyword query. In other words, a node is assigned a higher weight if it is more related to the keyword query.

2.3.2 Ranking Methods in Schema-Based KWS-R

Early work on keyword search over relational databases, like DBXplorer [1] and DISCOVER [17], mainly focus on the efficiency of the query system and simply rank the results by the size of MTJNTs. DISCOVER2 [15] first introduce the state-of-the-art Information Retrieval (IR) techniques into the ranking strategies on KWS-R, since IR-style ranking methods is widely used in document/web search. Later Liu et al. [26] suggest several sophisticated improvements to the ranking formula in DISCOVER2. More recently SPARK [27] improved the IR-ranking formula based on the idea of virtual document, which essentially consider JNTs as small documents. JNTs generated from the same CN are considered belong to a same collection of documents. Xu et al. [39] furthermore enhanced the formula by considering different relevance between the query and each relation.

Simple Way

Both DBXplorer and DISCOVER rank the results by the size of tuple trees. Let T be an MTJNT and let Q be a query, the score function is:

$$Score(T, Q) = \frac{1}{size(T)}$$

Intuitively, the smaller size of a tuple tree, the stronger connection of the tuples in the tuple tree.

IR-Style Method

Given a query and a collection of documents, IR systems assign a score for each document as an estimation of the document relevance to the given query. The widely used model to compute such a score is the *Vector Space Model*, in which each text (both documents and queries) is represented as a vector of terms, which could be a keyword or a phrase. A similarity (usually a dot product function) between a document vector D and a query vector Q can be computed as the ranking score.

DISCOVER2 [15] first applies IR-style ranking methods into the computation of ranking scores on keyword search over relational databases. They consider each column as a collection and each value in the column as a document. Specifically, let T be a MTJNT and $\{D_1, D_2, \cdots, D_m\}$ be all column values in T and let Q be the keyword query. DISCOVER2 uses a TF-IDF function to define the score, which is shown as follows:

$$Score(T, Q) = \frac{\sum_{D_i \in T} Score(D_i, Q)}{size(T)} \tag{2.1}$$

$$Score(D_i, Q) = \sum_{w \in D_i \cap Q} \frac{1 + \ln(1 + \ln(tf))}{(1 - s) + s\frac{dl}{avdl}} \ln \frac{N + 1}{df} \tag{2.2}$$

where, for each word w, tf is the frequency of w in D_i, df is the number of column values with word w in D_i's column, dl is the size of D_i in characters, $avdl$ is the average size of column values, N is the total number of values in D_i's column, and s is a constant (usually 0.2).

While DISCOVER2 straightforwardly applies IR-style methods, Liu et al. [26] propose four normalizations of the formulas, considering more the inherent structures of relational databases. For example, the intuition behind one of the normalizations using $size(T)$ in Equation (2.1) is that a MTJNT with more tuples tends to contain more terms and higher term frequencies. However, for a multi-keyword query, the relevant answers usually involves multiple tuples, each of which contains a subset of the query keywords. Such complex tuple trees deserve higher score than tuple trees with a single tuple, which contains a

small set of query keywords. Therefore instead of using the raw $size(T)$, they use the normalized $Nsize(T)$, defined as follows:

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

where $avgsize$ is the average size of tuple trees.

Both DISCOVER2 and Liu's system consider each attribute value as a document and all attribute values in the same column as a collection of documents. SPARK [27] however suggests to model the whole MTJNT as a document and all MTJNTs generated from same CN as a collection of documents. They use the similar formula to compute a score for each MTJNT. In SPARK's experiments, the new ranking method is shown to have a substantial improvement of the quality of search results.

Furthermore, Xu et al. [39] improve SPARK's method by introducing *query semantics* into ranking. DISCOVER2, Liu's system, and SPARK all use the TF-IDF scoring function, which is to compute the relevance between a keyword query and "documents" from a "document" collection. The only difference between these systems is the definition of a "document" and a "document" collection. The first two systems use an attribute value as a document and a column of values as a document collection, while SPARK use a MTJNT as a document and a CN as a document collection. Observe that in each case, there are many document collections, i.e., many columns and many CNs. The TF-IDF scoring function only compute a document's relative relevance to the query in the document collection it belongs to. However, different document collections have different level of importance to the query. In KWS-R, a final score of MTJNT should reflect the importance of the CN it belongs to. To achieve that, a concept of *query semantics* is defined as follows: The *semantics* of a query is the relation preference of the keywords. For example, given a query "*Hristidis*, *database*", which contains an author name "*Hristidis*" and a research area "*database*", it shows that the user wants to search a research paper about

database that was written by an author with "*Hristidis*" in his name. In this case, the semantics of the query is {`author`, `paper`}. It is easy to compute the semantics of a query, because one can simply use the IR-style method to find the most relevant relation for each keyword. Therefore, the system could go on to computes the relevance between each CNs and the semantics of the query, and integrate this relevance to the final scores of answers.

2.3.3 Discussion

To rank the results, a relevant score must be assigned for an answer , as we can see in all the methods. Different methods utilize different information provided by a MTJNT and the whole database. DISCOVER and DBXplorer use the property of a MTJNT. BANKS's method captures the different prestiges of each node and each edge, given their different positions in the network of the whole database graph. IR-Style method, on the other hand, looks into the actual text values of tuples in a MTJNT and compute the similarity between these text values and the keywords in the query. Moreover, the recent improvements of IR-Style method leverage the inherent structures in relational databases.

It is difficult to tell which method will return more relevant and interesting results to the user. Simple method might be already good enough, while more complicated method might have more stable results but also be more costly.

2.4 Empty Result Problem in Conventional Relational Databases

In database research, there has been many works on the empty result problem in general. However, most of them study the problem in the context of the conventional query paradigm, where a query is a SQL statement and an answer is a table of tuples; while we study the empty result problem in the context of keyword query, where a query is a set of keywords and an answer is a joining network of tuples. In this section, we discuss how these works are different from,

or related to ours.

Existing solutions to the empty result problem basically answer the following two questions:

1. Why does a particular query returns an empty set of answers?
2. How could the system return something useful to the user instead of nothing?

Early works [32, 33] in the literature mainly focus on the first question. For most solutions, the query is executed first. If no answer satisfies the query, the system goes back and looks for the reason, which could be a "wrong" query given by the user, for example, a query with an non-existing schema, or the database really does not have matching data. In our case (keyword search), the cause of empty result is much simpler, that is, there does not exist a series of foreign key references that could connect a set of tuples, which contain all keywords in the query. Moreover, it is less likely that the user would give a "wrong" query, because it is only a list of keywords. Therefore, given these two reasons, we are more interested in the second question.

Later on, people are interested in automatically [2, 5] or interactively [31, 21] returning useful results when no results is found for the query, which in effect answers the second question. Kießling et al. [21] propose an extension to the standard SQL, called Preference SQL, which basically extends SQL by providing the user an interface to specify *soft constraints*, for example, "`price AROUND 1000`." Compared to exact queries, i.e., using the standard SQL, such queries with soft constraints is less likely to generate empty answers. However, for our case, changing the query interface is not a desirable solution, because the simplicity of the query interface is the most important feature of keyword search, which should be preserved.

Alternatively, Mishra et al. [31] introduce a model that enables the user to interactively refine the query. In their system, if a query generates few or no answers, the user could relax one or more predicates on the fly to get more

answers. In keyword search, the only way to relax the query is to remove one or more keyword, however we are more interested in finding results that contains all keywords. In our case, it is possible that all keywords have matching tuples in the database, but the reason of empty answers is there exists no set of tuples that are connected via foreign key references and contains all keywords as a whole. Therefore, for now, we do not consider query relaxation in our work.

In Information Retrieval, the results of a document query are usually ranked and the most relevant documents are returned. Inspired by this, Agrawal et al. [2] and Chaudhuri et al. [5] adapt the ranking techniques from IR to handle both empty answers and many answers scenarios in relational database systems. In the case of empty answers, their system automatically generates a ranked list of *approximately matching* tuples. To achieve that, first, a *similarity* is defined between any tuple and the query. It can be based on vector space model [2] or probabilistic model [5]. Then, for a SQL query that generates empty answers, the system retrieves and finds tuples that are most relevant to the query, according to their similarities. Preprocessing and index are used to speed up the computation of similarity, and query workload is leveraged in similarity model to improve the quality of the results. To our knowledge, their work is most relevant to ours. However, in addition to the different query paradigms as mentioned above (SQL vs keyword), our work is also distinct from theirs in other ways. First, we consider an answer in the final results should contain all the keyword in the query, while in [2, 5], a tuple in the results may not satisfy all the predicates in the query, as long as it has very high similarity to the query. Second, we define not only the similarity between any tuple (actually, in our case, it is a joining tuple graph) and the query, but also the similarity between any two tuples.

2.5 Tuple Similarity

Another related work is tuple similarity problem, which has been intensively studied in many areas, like data cleaning, or data integration. These works [10] attempt to match two tuples from same or different databases that actually refer to the same real world object. The existing solutions can be divided into two categories: *learning-based* approaches and *distance-based* approaches. Learning-bases approaches use probabilistic models or machine learning techniques to "learn" how to match the tuples. However, these methods must rely on good training data. In contrast, distance-based approaches only rely on distance metrics to match similar tuples. Any tuple similarity measure can be used in our solution, however a typical database does not always have a good training data for tuple similarity, we are more interested in the *distance-based* approach, which can always be applied to a database.

The basic idea of the distance-based method is that, given a tuple, it first computes the similarity between this tuple and others, based on some distance function, and then defines a matching threshold to find the most similar tuples. Many similarity metrics have been developed to measure the similarity of two tuples. There are mainly two categories: *edit-based* metrics [13, 19] and *token-based* metrics [36]. Edit-based metrics, like edit distance and q-gram distance, work very well for typographical errors. However these metrics often fail to deal with rearrangement of words. Token-based metrics compensate for this kind of problem. Cohen [7] introduced WHIRL system to first adopt IR technique, the cosine similarity with TF/IDF weighting scheme, to measure the similarity of two tuples. Later Gravano et al. [12] extended the TF/IDF metric by using q-grams instead of words, in order to handle spelling errors. We adopt Cohen's metric in our problem, because for our problem, we are interested in finding similar tuples that may contain several same words; these tuples possibly provide same information to users. Handling spelling errors is costly and make the similarity function more complex. Therefore, we simply assume no spelling

errors in the database.

Two tuples to be matched may or may not have same structure. For tuples with same structure, we can just compare each individual field in two tuples and combine all the similarity scores to obtain a total score for the whole tuples. However, when two tuples do not have same structure, it is impossible to compute the similarity in this way. A simple approach is that we just consider the whole tuple as a single field, and then use appropriate metrics to compute the similarity. Since we adopt the cosine similarity in Cohen [7], we do not lose much information in this approach, because this metric only rely on the word frequencies but not the positions of words.

Chapter 3

Empty Result Problem In KWS-R

The *empty result problem* in KWS-R presents the scenario that we cannot find an answer which contains all the keywords in the query. The limitation of the graph model of relational databases is that there is only one type of connection (foreign key reference) to represent the relationship between tuples. If two keyword tuples cannot be connected via a series of foreign key references, there is no way that the two tuples appear in the same answer. Therefore, no answer will be returned for a query, if the database does not have a set of keyword tuples that (1) cover all the keywords and (2) are connected via foreign key references. Instead of returning nothing, it is better to return something that will be meaningful to the users in some way. Both Graph-Based KWS-R and Schema-Based KWS-R have the empty result problem, since they both use the graph model and consider only foreign key references. In this work, we focus on Schema-Based KWS-R. For Graph-Based KWS-R, the solution in this paper cannot be applied directly; therefore, it remains as a further work.

We consider only AND semantics for keyword search, which must return answers containing all the keywords. There is also OR semantics, which returns

answers that contain at least one but not necessarily all the keywords. OR semantics can be regarded as a trivial solution to the empty result problem, however it is more interesting and more challenging to consider AND semantics while solving this problem.

Our solution is based on tuple similarity. The basic idea is that a new relationship is added for tuples in the database, in addition to foreign key reference, such that when tuples cannot be connected by foreign key references, they can be connected by the new relationship. We introduce *Similarity Connection* for two tuples as follows.

Definition 6 (Similarity Connection)**.** *Given two tuples $t_i, t_j \in V$, where V is the set of nodes in G, t_i is connected to t_j by similarity, iff the similarity score w of t_i and t_j is larger than a predefined threshold θ.*

Note that the similarity connection is symmetrical. If t_i is connected to t_j with a similarity score w, t_j is also connected to t_i with the same similarity score w.

In the rest of this section, we first introduce the answer model used in our solution, then we define the ranking function for the results. We also describe the similarity measure that are adopted in the solution and show a modified inverted index that accelerates the search of similar tuples. Finally, we propose an algorithm that find the answers to the empty result problem efficiently.

3.1 Answer Model

Generally speaking, for a user who queries a database by keywords, he or she expects a result that satisfy the following two criteria:

1. The result contains all the keywords in the query.
2. The result is a single integrated piece of information. For example, it could be a graph of tuples, which are connected by foreign key reference;

or it could be a set of tuples, which are not connected, but these tuples together provide some information.

The first criterion is straightforward. The second one means that the result is not several isolated pieces of information. For example, for a keyword query "`database Agrawal Chaudhuri`" in DBLP, the user is likely to look for one single paper about database that is co-authored by Agrawal and Chaudhuri, or two separate papers, one is from Agrawal and the other is from Chaudhuri, but they are both about database. However, the user is less likely to expect a paper about database from Agrawal and another paper from Chaudhuri that is totally unrelated to Agrawal's paper. Users are interested in the information that has connections.

Therefore, to consider some results that are still interesting to the user, even if the system fails to find an answer in the traditional way, we still have to follow the two criteria.

Remember in Schema-Based KWS-R, $G_{ES}(Q)$ denotes a expanded schemas graph. In the empty result problem, for a $G_{ES}(Q)$, there is no actual candidate network (containing all the keywords) but only a set of partial candidate networks, which also cannot generate MTJNTs but only tuple trees that contain a partial set of keywords. We formally define *Partial Candidate Network* and *Minimal Partial JNT* as follows.

Definition 7 (Partial Candidate Network (PCN))**.** *A partial candidate network is a subgraph of $G_{ES}(Q)$ that contains at least one but not all the keywords in the query, and it is impossible to remove any node from it and make the remainder contain the same set of keywords as before.*

Definition 8 (Minimal Partial Joining Network of Tuples (MPJNT))**.** *A minimal partial JNT is a JNT that contains at least one but not all the keywords in the query, and it is impossible to remove any node from it and make the remainder contain the same set of keywords as before.*

Similar to CNs and MTJNTs, MPJNTs are generated by evaluating PCNs, and PCNs can be considered as the projection of MPJNTs onto the expanded schema. Given a list of all MPJNTs, a subset of MPJNTs will be interesting to the user, if they together contain all keywords and they are also connected in some way. Both MPJNTs and JNTs can be connected by similar tuples or foreign key references. Therefore, we define the connection of two JNTs.

Definition 9 (Connection of Two JNTs)**.** *A JNT τ_i is connected to a JNT τ_j iff there is a tuple t_i in τ_i that is connected to a tuple t_j in τ_j by similarity, or there is t_i in τ_i that is connected to a tuple t_j in τ_j by a foreign key reference.*

Hence, an answer model to the empty result problem is defined as follows:

Definition 10 (Answer to Empty Result Problem)**.** *An answer $\mathcal{A}$ to the empty result problem is a connected network of MPJNTs and JNTs that contains all the keywords in the query, and it is impossible to remove any MPJNT from $\mathcal{A}$ and make the remainder still contain all the keywords.*

Note that in an answer $\mathcal{A}$, two MPJNTs can be directly connected, or can be connected through a JNT that does not contain any keywords.

3.2 Ranking Methods

A ranking method is required by the system to return the most interesting answers first, as the number of possible answers could be very large. In Section 2.3, we discussed ranking methods that are proposed in the literature for KWS-R; however, these methods cannot be directly used in the empty result problem, because the answer models in the two problems are different.

For the empty result problem, we have two main components in the answer model to consider for a ranking function: (1) MPJNTs and (2) the edges connecting MPJNTs. For MPJNTs and JNTs, we consider them as a small document and adapt the IR-Style scoring function in Xu et al. [39]. Hence, each MPJNT and JNT has a score $Score(\tau_i)$ with regard to the keyword query.

The scores of the edges between MPJNTs are more complicated. For the sake of simplicity, we consider an answer with only two MPJNTs τ_1 and τ_2. For more than 2 MPJNTs, the connection of any two connected MPJNTs would be one of the following ways. There are in total three ways that the two MPJNTs are connected:

1. τ_1 and τ_2 are directly connected by a similarity edge.
2. τ_1 is connected to a JNT τ_3 by a similarity edge , and τ_2 is connected to τ_3 by a foreign key edge.
3. τ_1 and τ_2 are both connected to a JNT τ_3 by similarity edges.

We have two types of edges: similarity edges and foreign key edges. These two edges are not equally important in an answer; foreign key edge is stronger than similarity edge. The weight of the similarity edge is just the similarity score of the two corresponding tuples. A value between 0 and 1 (Details of the similarity measure of two tuples are discussed in the next section). Therefore, we simply assign a same weight of 1 to every foreign key edge. As a result, 2) can be considered as a special case of 3).

For an answer $\mathcal{A}$, let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ denote all MPJNTs and JNTs, and let $\mathcal{E} = \{e_1, e_2, \cdots, e_m\}$ denote all similarity edges and foreign key edges that connect MPJNTs and JNTs. Hence, we have the following scoring function:

$$Score(\mathcal{A}, Q) = \frac{\sum_{e_i \in \mathcal{E}} (Score(\tau_{i,l}) + Score(\tau_{i,r})) \times Sim(e_i)}{N} \tag{3.1}$$

where $\tau_{i,l}$ and $\tau_{i,r}$ are the two JNTs or MPJNTs that are connected by e_i, $Sim(e_i)$ is the similarity score of two tuples that connected by e_i, and N is the total number of edges in $\mathcal{A}$ including foreign key edges inside MPJNTs and JNTs.

3.3 Similarity Measure

In the literature, there are many ways of defining the similarity of two tuples in the database. Any similarity definition can be used in our solution, however a different similarity definition would result in different answers. Given a specific database, how to choose a good similarity definition that makes the system perform efficiently and generates good results also remain as a further work. In this paper, we use the *cosine similarity* with TF-IDF weighting in our work, because of two reasons: (1) Many works [15, 27, 39] about ranking results of Schema-Based KWS-R use IR-style ranking formula, which is also based on TF-IDF weighting; (2) the cosine similarity of two tuples is independent of the schema of the two tuples, so even two tuples from different tables can have a similarity score.

In *cosine similarity*, we consider a tuple in the database as a small document, which consists of the values of attributes in the tuple. Therefore, the database is considered as a collection of small documents. In the *vector space model*, tuples in the database are represented as term vectors. For example, we have a tuple as follows in the database:

title	**authors**
`Keyword search in relational databases: a survey`	`Jeffrey Xu Yu, Lu Qin, Lijun Chang`

This tuple is considered as a small document by combining the values of attribute "title" and "authors." Thus, this tuple can be represented by the vector as follows.

```
{keyword:0.083, search:0.083, relational:0.083, databases:0.083,
   survey:0.083, jeffrey:0.083, xu:0.083, yu:0.083, lu:0.083,
            qin:0.083, linjun:0.083, chang:0.083 }
```

Each component of the vector is actually a key/value pair. Keys are the terms in the tuple, and the value of each key is the weight of the term. For each vector,

the number of dimensions is the total number of terms in all the documents. However, we omit terms that have zero value, because vectors will be multiplied. There are many methods to define weights of terms. The above example gives a simple weighting method, the *term frequency*, which is the occurrence of the term divided by the total number of terms in the tuple. Since every term in this example only occur once, their values are the same, $1/12 = 0.083$. The most popular weighting method is *TF-IDF weighting*, which combines the *term frequency* and the *inverse document frequency*.

Therefore, for two tuples t_i and t_j, assuming their term vectors are $V(t_i)$ and $V(t_j)$, the similarity of t_i and t_j is defined as:

$$Sim(t_i, t_j) = \frac{V(t_i) \cdot V(t_j)}{|V(t_i)||V(t_j)|} \tag{3.2}$$

$Sim(t_i, t_j)$ is a cosine value, and $0 \leq sim(t_i, t_j) \leq 1$. When $Sim(t_i, t_j) = 1$, t_i and t_j are the same; when $Sim(t_i, t_j) = 0$, t_i and t_j are totally different in terms of words, meaning no overlap of the two term vectors.

The effect of the denominator of Equation 3.2 is the length normalization of the vectors $V(t_i)$ and $V(t_j)$. Therefore, we can rewrite Equation 3.2 as:

$$Sim(t_i, t_j) = v(t_i) \cdot v(t_j) \tag{3.3}$$

where

$$v(t_i) = \frac{V(t_i)}{|V(t_i)|},\ v(t_j) = \frac{V(t_j)}{|V(t_j)|}$$

$v(t_i)$ and $v(t_j)$ are called the unit vector of $V(t_i)$ and $V(t_j)$. Hence the similarity of two tuples is actually the dot product of their unit vector.

3.4 Similarity Index

It is critical to find similar tuples efficiently in our solution. Essentially, there are two phases in searching similar tuples for a given tuple: (1) computing the

similarity score between the query tuple and other tuples, and (2) ranking the result tuples according to the similarity scores. Because of the similarity measure we choose, the two phases are very similar to keyword search for documents in Information Retrieval [28]. The latter is basically to retrieve the top-k documents ranked by the similarity score between the keyword query and documents. In our case, the query tuple is considered as a keyword query and the rest of tuples in the database are the documents to be retrieved. Therefore, we can use the technique in keyword search for documents to find similar tuples efficiently.

The key idea of searching similar tuples efficiently is to compute the similarity score while traversing the inverted index. For a query term, an inverted index can quickly return a list of tuples that contain this term. A tuple is simply stored by its ID, which is called a *posting* in the inverted index. The list of tuples returned for a term is also called the *postings list* of this term.

Algorithm 1 shows the procedure that utilizes an inverted index to efficiently find the similar tuples. We walk through the postings in the inverted index for the terms in the query q, accumulating the total score for each document. Specifically, for each term t in the query q, we first fetch the postings list for t using the inverted index. For each tuple d in posting list, we multiply the values of term t in q and d. This product is added to the score of tuple d. Finally, given these scores, the K highest-scoring tuples are returned.

Algorithm 1 SimTuple(q)

```
1: Scores[N] = 0 // record the scores of each tuple
2: for each query term t in q do
3:    w_{q,t} = value of term t in the vector of q
4:    fetch postings list for t
5:    for each tuple d in postings list do
6:       w_{d,t} = value of term t in the vector of d
7:       wt_{q,d} = w_{q,t} × w_{d,t}
8:       Scores[d] += wt_{q,d}
9:    end for
10: end for
11: return Top K components of Scores[]
```

In Algorithm 1, we still need to compute the product of the values of each

	1 (0.08)	2 (0.13)	3 (0.25)	...
1 (0.08)	0.0064	0..0104	0.02	...
2 (0.13)	-	0.0169	0.0325	...
3 (0.25)	-	-	0.0625	...
...	...	...	...	...

Table 3.1: Product Table of Term "`database`"

term t in q and d (line 7). The multiplication is expensive, especially the multiplication of two floating point values. To eliminate the multiplication and improve the algorithm, we consider to pre-compute the products of every pair of all possible values for each term, and store these results as a product table. For example, if term "`database`" has values: 0.08, 0.13, 0.25, 0.37, ..., each of which appears in one or more vectors. In other words, we list all the distinct values for term "`database`" that appears in all vectors. We can construct a product table of term "`database`" as shown in Table 3.1.

We also need to store the indexes of corresponding values in the inverted index. For example, in the postings list of term "`database`," if a vector has value of 0.13 for this term, we store the column of this value in the product table with this vector.

Given product tables and the modified inverted index, we need to change Algorithm 1 to leverage these auxiliaries. The new version is shown in Algorithm 2. In line 3 and line 6, indexes of values are retrieved instead of values themselves. In line 7, we lookup the product table of term t for the product, instead of computing it on the fly.

Algorithm 2 can be further improved by compressing the product tables. Note that Algorithm 2 is only useful if Products($\cdot$) could retrieve the result in the corresponding product table faster than the multiplication operation of two floating point values. Therefore, the product tables must be in the memory. However, since every term that appears in the database requires a product table, and each term may have many different values, in the worst case, the number of product tables and the size of product table can be very large so that they cannot be fitted in the memory.

Algorithm 2 SimTuplePT(q)

Require: Products(*term*,i_1,i_2), which retrieve the item of index (i_1,i_2) in *term*'s product table.

```
1: Scores[N] = 0 // record the scores of each tuples
2: for each query term t in q do
3:    i_{q,t} = index of term t's value in the vector of q
4:    fetch postings list for t
5:    for each tuple d in postings list do
6:       i_{d,t} = index of term t's value in the vector of d
7:       wt_{q,d} = Products(t, i_{q,t}, i_{d,t})
8:       Scores[d] += wt_{q,d}
9:    end for
10: end for
11: return Top K components of Scores[]
```

However, for the purpose of ranking the tuples, we are really interested in the relative, rather than absolute, scores of the tuples in the database. The product table thus can be compressed by reducing the number of distinct values. In other words, we break the range of the values into a small number of intervals, and for each pair of intervals, we compute the average of products of value pairs that fall in the two intervals. The product table only contain the average product for every pair of intervals. In this way, the size of product table could be largely reduced.

3.5 Query Processing

To find answers in the empty result situation, we need to first obtain all the MPJNTs, and then find similar tuples among these MPJNT to connect them. For the first phase, we modified algorithms used in Schema-Based KWS-R [17, 29]. For the second phase, we introduce two new algorithms to search for answers.

3.5.1 MPJNT Generation Algorithm

In Section 2.2.2, we discuss two phases to find MTJNTs in Schema-Based KWS-R: CN generation and CN evaluation. Similarly, there are two phases to find

MPJNTs: PCN generation and PCN evaluation. Since CN evaluation and PCN evaluation are essentially the same process, which is evaluating a graph of expanded schemas to generate a graph of tuples, we use the same algorithm from CN evaluation for PCN evaluation. However, for PCN generation, we cannot use the same algorithm [29] from CN generation, because a CN contains all the keywords while a PCN does not. Fortunately, it is easy to modify the algorithm to generate PCNs. To do that, we only need to remove its the acceptance condition, which is that a CN contains all the keywords. (Actually, the acceptance condition would never be satisfied, because we are in the empty result situation.) At the end, the set of subgraphs left, which have not been pruned, is the set of final PCNs.

3.5.2 ExpandTuple Algorithm

After all MPJNTs are generated, we begin to search for combinations of MPJNTs by using similar tuples. To do that, we introduce *ExpandTuple*, as shown in Algorithm 3. A list of MPJNTs is denoted by $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$, which are ranked by the number of keywords they contain. Specifically, τ_1 contains the largest number of keywords, while τ_n contains the least. A heap H is also maintained to track the ranking of the results. Therefore, the key of H is the ranking score of the results computed by Function 3.1.

In order to find connections between MPJNT, for each MPJNT τ_i from $\mathcal{T}$, we attach to τ_i the most similar tuple of each tuple in τ_i (line 4 to line 9). Intuitively, it is like expanding the MPJNT by similarity connection to cover more tuples in the database. After one expansion, we check the following three things:

1. If the expanded τ_i already contains all the keywords, it is removed from $\mathcal{T}$ and added to H. (line 10 to line 13)

2. Otherwise, if τ_i can be combined with some MPJNTs in $\mathcal{T}$, we combine τ_i with the one that contains largest number of keywords. Two MPJNTs can

be combined iff they have common tuples, the same tuples that appear in both MPJNTs. (Two MPJNTs originally must have no common tuples, but after the expansion of one MPJNT, they might have common tuples.) If the newly combined MPJNT contains all the keywords, it is added to H; otherwise, it is added to $\mathcal{T}$. (line 14 to line 24)

3. If τ_i in $\mathcal{T}$ cannot be combined any MPJNT in $\mathcal{T}$, then for each newly attached similar tuples, we search for foreign key connection to all the tuples in $\mathcal{T}$. Connect τ_i to the MPJNT with largest number of keywords if there is any. If the newly connected MPJNT contains all the keywords, it is added to H; otherwise, it is added to $\mathcal{T}$. (line 25 to line 36)

In step 3, the length of the foreign key connection is limited by a predefined threshold γ. In our experiment, we set $\gamma = 1$, because when $\gamma > 1$, the algorithm is very slow. It could be a future work to improve the algorithm when $\gamma > 1$.

If $\mathcal{T}$ is not empty, we repeat the expansion process again. For each tuple, the next most similar tuple is attached. We keep on expanding the tuple trees in $\mathcal{T}$ iteratively until $\mathcal{T}$ is empty.

Whenever an answer is found, we need to minimize it before adding it to H, line 11, 18 and 29 in Algorithm 3. During expansions of the tuple tree, many similar tuples are attached to it. Some similar tuples contribute to the answer, because they contain keywords that the original tuple tree do not have, or they are the common tuples of two or more MPJNTs. However, other similar tuples are not that important, so those similar tuples should be removed from the answer before it is added to H. Therefore, we minimize answers by removing these tuples.

To analyze the complexity of ExpandTuple, let the average number of tuples in all the MPJNTs be m, and the number of MPJNTs be $|\mathcal{T}|$. In experiments, we pre-compute the similar tuples for each tuple in the database using SimTuplePT, therefore line 5 costs constant time. Thus, the time complexity for each expansion is $O(m)$. In both line 14 and line 25, in the worst case, we

Algorithm 3 ExpandTuple($\mathcal{T}$)

Require: $\mathcal{T}$: List of MPJNTs ranked in descending order of the number keywords they contain.
Require: H: the result heap

```
1: H ← ∅
2: while T ≠ ∅ do
3:   τ_i ← first MPJNT in T
4:   for each tuple t in τ_i do
5:     t_nb ← next similar tuple returned by SimTuplePT(t)
6:     if t_nb has not been attached to τ_i then
7:       τ_i ← τ_i ⇝ t_nb
8:     end if
9:   end for
10:  if τ_i contains all keywords then
11:    minimize τ_i
12:    H ← H ∪ τ_i
13:    T ← T − τ_i
14:  else if τ_i can be combined with at least one MPJNT in the rest of T then
15:    τ_j ← the one with largest # of keywords from all MPJNTs that can be combined
16:    combine τ_i and τ_j into τ'_i
17:    if τ'_i contains all keywords then
18:      minimize τ'_i
19:      H ← H ∪ τ'_i
20:    else
21:      T ← T + τ'_i
22:    end if
23:    T ← T − τ_i
24:    T ← T − τ_j
25:  else if τ_i can be connected to at least one MPJNT in the rest of T then
26:    τ_j ← the one with largest # of keywords from all MPJNTs that can be connected
27:    connect τ_i and τ_j into τ'_i
28:    if τ'_i contains all keywords then
29:      minimize τ'_i
30:      H ← H ∪ τ'_i
31:    else
32:      T ← T + τ'_i
33:    end if
34:    T ← T − τ_i
35:    T ← T − τ_j
36:  end if
37: end while
38: return H
```

need to check each tuple in τ_i with all tuples in the rest of $\mathcal{T}$. Therefore, the time complexity for the two parts is $O(m \cdot (|\mathcal{T}| - 1) \cdot m)$. And the minimization of MPJNT costs $O(m)$. The total time complexity of ExpandTuple is $O(|\mathcal{T}| \cdot (m + m^2 \cdot (|\mathcal{T}| - 1) + m))$, which is $O((m \cdot |\mathcal{T}|)^2)$. Thus, ExpandTuple could be slow when $|\mathcal{T}|$ is large. Unfortunately, $|\mathcal{T}|$ is usually very large, and it is exponential growth with the number of keywords. Therefore, we design the second algorithm *ProgressiveExpandTuple* to improve the performance.

3.5.3 ProgressiveExpandTuple Algorithm

ProgressiveExpandTuple is a progressive algorithm, which means it returns results progressively. Instead of searching for results after all MPJNTs are found, it starts searching as soon as possible and return the results whenever it finds one. It also updates the final results according to the scores of old and new results.

The main idea is that we add this procedure into the PCN evaluation algorithm, in which it is triggered whenever a new MPJNT is found. As shown in Algorithm 4, it accepts a heap $\mathcal{T}$ of MPJNTs as a parameter, which contains all the MPJNTs that are found so far and have not been processed by ExpandTuple. If $\mathcal{T}$ contains all the keywords, it processes $\mathcal{T}$ by sending it to ExpandTuple. Then it adds the results of ExpandTuple to the result heap, and present it to users. Note that $\mathcal{T}$ will be cleared when it is processed by ExpandTuple. ProgressiveExpandTuple will be triggered again when a new MPJNT is found in the PCN evaluation algorithm.

Note that the results found by ProgressiveExpandTuple are different from that found by ExpandTuple. ProgressiveExpandTuple finds results from a small set of MPJNTs, while ExpandTuple finds results from all MPJNTs. The former is locally optimized, and the latter is globally optimized. As shown in the experiments, ExpandTuple provides results with better quality than ProgressiveExpandTuple does, however, ProgressiveExpandTuple returns results faster than ExpandTuple does. It is the trade off between the quality and the speed.

Algorithm 4 ProgressiveExpandTuple($\mathcal{T}$)

Require: $\mathcal{T}$: A heap of MPJNTs ranked in descending order of the number keywords they contain. It contains MPJNTs that are found so far and have not been processed by *ExpandTuple*.
Require: H: the result heap
1: **if** $\mathcal{T}$ contains all keywords **then**
2: $H \leftarrow$ ExpandTuple($\mathcal{T}$)
3: **end if**
4: Present H

Chapter 4

Experiments

The effectiveness evaluation of keyword search over relational databases remains as an open challenge [38]. In the literature, several works [39, 27, 26, 15] mainly focus on the effectiveness of KWS-R. However, there are many problems in their experiments of effectiveness evaluations, as pointed out in [38]. For example, first, queries in the experiments are generally formulated by the authors themselves, but self-authored queries usually have a strong potential bias, since it is very easy to choose queries that are favorable to their own algorithms than others'. Second, query sets are not reusable such that it is difficult to compare the results from experiments to experiments.

Recently, Coffman et al. [6] propose a framework for evaluating keyword search over relational databases. Essentially, they provide datasets, queries and a set of relevant results as the ground truth for evaluation, and they follow the best practice from the establishment of frameworks like TREC. However, we cannot directly use Coffman et al.'s framework in our solution, because we need queries that do not return answers in the traditional keyword search in relational databases. Nevertheless, we reuse the ground truth that are provided in the framework. We generate empty result queries from the original queries in the framework, such that the ground truth of the original queries can be used to assess the relevance of the results for the empty result queries. Therefore,

a result for an empty result query is relevant iff the result contains all the information that is in the relevant answer for the original query of the empty result query. More details are discussed later.

Our experiment evaluates the effectiveness and the performance of the system. For the effectiveness, we want to find out how the results are relevant to the query. For the performance, we want to show what is the overhead to use our system to find answers when the query generates empty results originally.

4.1 Data Set

Two datasets from the framework [6] are used, IMDb and Wikipedia. The size of IMDb dataset is 516MB, and it has 6 relations and 1,673,074 tuples in total. The size of Wikipedia dataset is 550MB, and it has 6 relations and 206,318 tuples in total.

4.2 Query Set

As mentioned above, we reuse the ground truth in the framework [6], and therefore we generate our query set from the queries in the framework [6]. For a original query and its corresponding relevant result, our target is to find a new query with the intent that is similar to the original intent as close as possible, and it also returns empty results in the traditional KWS-R system. The basic idea is to replace one keyword in the old query with a new keyword so that the new query meets those goals.

At first, we wrote a program, based on simple heuristics, that automatically generates empty queries from original ones. Specifically, for an original query $Q = \{k_1, k_2, \ldots, k_m\}$, it first gets a relevant result in the ground truth. Next, it randomly picks a keyword k_i in Q, and find a tuple t_s that is similar to the tuple in the result that contains k_i. Then it searches for a keyword k_j in t_s such that all tuples that contain k_j cannot form a new result with the rest of the

tuples in the original result, because either they are not connected, or the size of the tuple tree exceeds the size limit of results. If k_j is found, then Q with k_i being replaced by k_j is the new query. Otherwise, a new keyword k_i in Q is picked, and the process is repeated until a satisfied k_j is found.

There are two problem with this program. First, it is not efficient. It takes long time to generate one query. Second, most importantly, the new replacement keyword is usually not related to the old keyword or the rest of keywords in the query. Therefore, the intent of the new query is not guaranteed to be similar to the original intent.

Finally, we manually generate new queries by replacing keywords by their synonyms until the new queries are actually empty queries. It is more time-consuming, but it guarantees that the new queries have the similar intents with the original ones', so that it is appropriate to use the ground truth from the framework in our experiments.

We generate 50 queries for each data set. In the framework [6], there are 50 queries for each IMDb and Wikipedia datasets. However, some queries contain only one keyword, and we cannot generate new queries from them, because for our system, queries should contain at least two keywords. Therefore, we generate more than one new queries for some original queries. 50 queries for each databases are listed in Appendix 6.1.

4.3 Metrics

To measure the effectiveness of our system, we adopt two metrics used in the previous study [27]:

1. **Number of top-1 answers that are relevant**. We run all the queries, and take the top-1 answer for each query, then compare it with the relevant answers of that query. It is the number of queries for which the first answer is relevant.

2. **Mean reciprocal rank**. For each query, we select the most relevant

answer in its results list. If the position of the most relevant answer is r, then the reciprocal rank of that query is: $1/r$. The mean reciprocal rank of all the queries $\mathcal{Q}$ is:

$$\frac{1}{|\mathcal{Q}|}\sum_{i=1}^{\mathcal{Q}}\frac{1}{r_i}$$

To calculate both metrics, we retrieve the top 10 results for our system. The first metric measures the quality of top 1 answers while the second metric measures the quality of top 10 answers.

4.4 Implementation

Since our work is based on Schema-Based KWS-R system, we implement it by ourself as described in Section 2.2.2. CN generation process is implemented according to a pre-order traversal algorithm proposed in Markowetz et al. [29]. CN evaluation process is implemented according to DISCOVER [17]. We use Xu et al. [39]'s ranking method in our Schema-Based KWS-R system, as it is the most advanced IR-style ranking method for Schema-Based KWS-R.

We assume that the databases are not frequently updated (inserting or deleting tuples), in which case similar tuples for each tuple are not changed frequently. Therefore, we pre-compute top 10 similar tuples for each tuples in the databases. Instead of searching for similar tuples on the fly, we can fetch similar tuples very quickly by just looking up a pre-computed table. However, it is very expensive to pre-compute the similar tuples for the entire database in terms of both time and space. We use about 25.5 hours to compute the table for IMDb dataset and about 18 hours for Wikipedia dataset. The time depends on both the number of tuples and the size of tuples while the size of pre-computed table only depends on the number of tuples. For IMDb, the pre-computed table is 149MB; for Wikipedia, it is 26MB. Therefore, our solution is not flexible for databases that are updated frequently. Incremental updates on the indexes remains as a further work.

We implements both ExpandTuple and ProgressiveExpandTuple algorithms. In both cases, we use the same ranking methods discussed in Section 3.2 for ranking the final results.

Our experiments setup is a PC with Intel Core Duo CPU 2.33 Ghz and 4 GB memory, running 32-bit Window 7 Professional Operating System. We use PostgreSQL 8.4 as our database management system. Our system is implemented in Java and communicated with PostgreSQL through JDBC interface.

4.5 Results

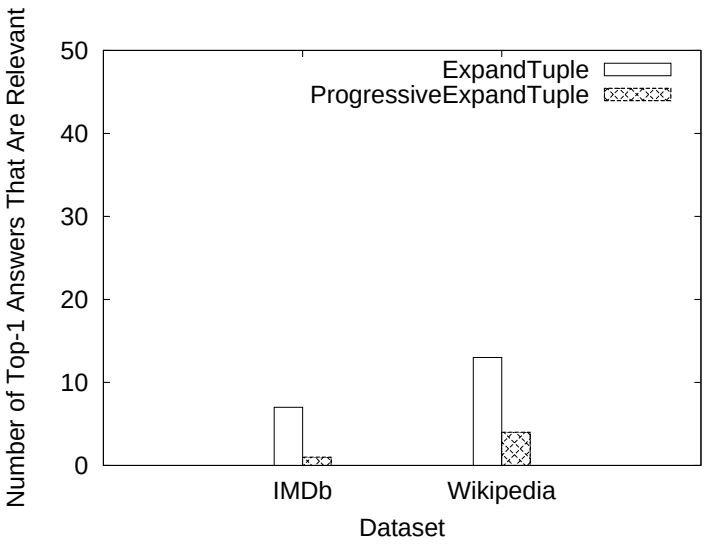


Figure 4.1: Comparison of Number of top-1 answers that are relevant for all queries for each dataset. Higher bars are better.

Figure 4.1 and Figure 4.2 summarize the effectiveness of our system. Figure 4.1 shows the number of top-1 answers that are relevant for all queries for each dataset. When using ExpandTuple algorithm, for IMDb dataset, there are 7 of 50 top-1 answers that are relevant, while there are 13 of 50 top-1 answers for Wikipedia dataset. When using ProgressiveExpandTuple, IMDb dataset

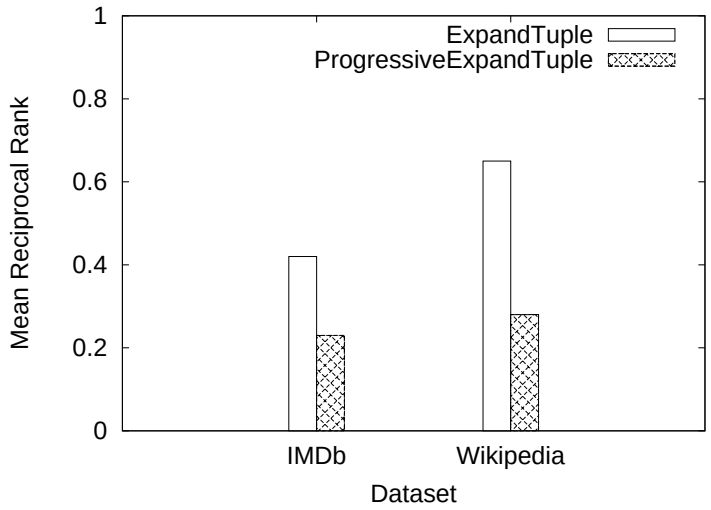


Figure 4.2: Comparison of Mean reciprocal rank for all queries for each dataset. Higher bars are better.

only has 1 of 50 relevant top-1 answer, and Wikipedia dataset has 4 of 50 relevant top-1 answers. Therefore, ExpandTuple provides better results than ProgressiveExpandTuple does.

There is no existing system that solves the empty result problem for KWS-R, we don't have other systems to compare. However, in [6], there is a comparison of top-1 relevant answers for all traditional KWS-R system, and the average number is 28. Although it is based on different query set and dataset, it can be used as a reference. Our system is not as effective as traditional KWS-R systems, but it indeed is capable to return the most relevant answers for some queries.

Figure 4.2 shows the mean reciprocal rank for all queries for each dataset. With ExpandTuple, for IMDb dataset, the mean reciprocal rank is 0.42, while for Wikipedia dataset, it is 0.65. It shows our system is capable to return relevant answers. ProgressiveExpandTuple again has lower mean reciprocal ranks than ExpandTuple in both datasets.

In both figures, the metrics of Wikipedia dataset are higher than that of IMDb dataset. The difference, however, depends on various factors, such as query set, size of the dataset or schema of the database. We use different query sets for two datasets, therefore, it is hard to tell the real reason. Nevertheless, one possible reason is that for Wikipedia, tuples tend to contain more text than that in IMDb, and our similarity connection is more useful when tuples contain more text.

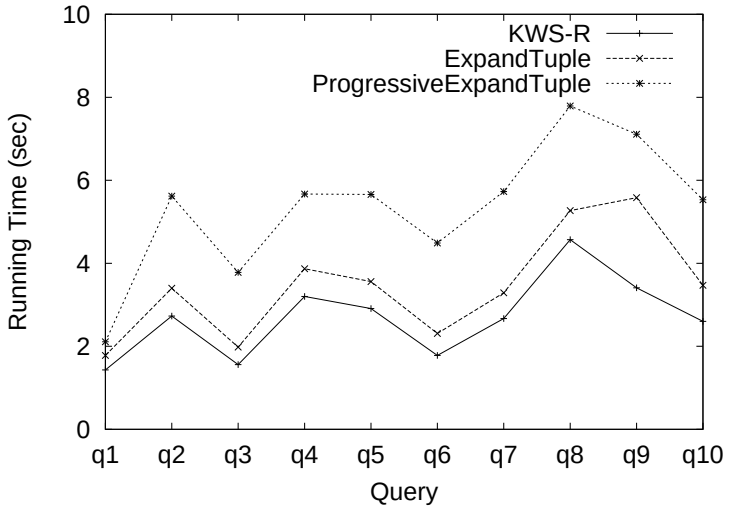


Figure 4.3: Comparison of Running Time of Traditional KWS-R and Empty Result KWS-R

For comparison, we also implement the traditional KWS-R. Figure 4.3 shows the performance of our system. When evaluating the performance, our goal is to show how much overhead the user would get when using our system to find answers when empty results is found originally. For simplicity, we randomly choose 10 queries out of 50 to show in Figure 4.3. They are good representation of overall performance. For each query, the running time is the time from the beginning to when the very first result is returned. The traditional KWS-R and empty result KWS-R with ExpandTuple algorithm do not progressively

return answers, but empty result KWS-R with ProgressiveExpandTuple algorithm does. As shown in Figure 4.3, empty result KWS-R with ExpandTuple take time of 1.5 to 2.5 times more than traditional KWS-R does. In other words, when no result is returned for the original query, the user need to wait 0.5 to 1.5 times more time before he or she could get an answer from our system. It is not very long, but ProgressiveExpandTuple reduces the waiting time of the first answer as shown in Figure 4.3. Empty result KWS-R with ProgressiveExpandTuple algorithm requires the user to wait much less to get answers. ProgressiveExpandTuple improves the user experience in items of response time.

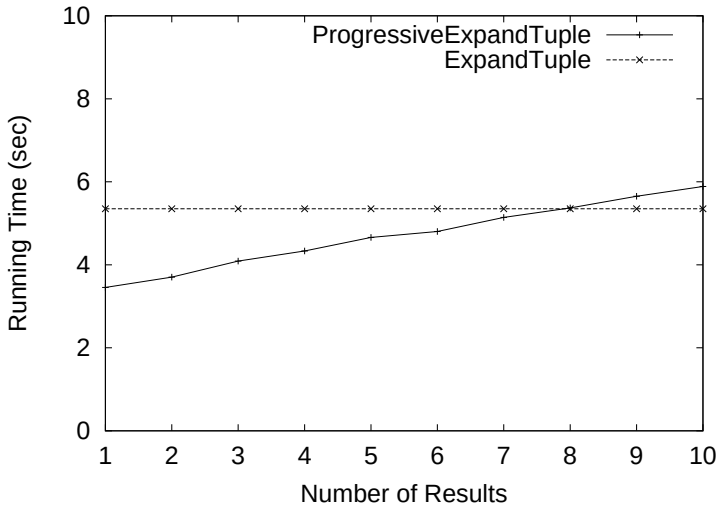


Figure 4.4: Comparison of ExpandTuple and ProgressiveExpandTuple

Figure 4.4 compares ExpandTuple and ProgressiveExpandTuple, and shows how ProgressiveExpandTuple algorithm reduces the response time and improves the user experience. ExpandTuple does not return answers until it finds all of them, therefore the time when the user gets the first answer is the same as the time he or she gets the first 10 answers. ProgressiveExpandTuple, on the other hand, progressively returns answers, therefore it starts searching results as soon

as possible, and returns the results whenever it finds them. In this way, as shown in Figure 4.4, the time the user gets the first answer is much earlier that the previous approach.

However, we also notice that the time the user gets the fist 10 answers is actually greater in ProgressiveExpandTuple than in ExpandTuple. One of the explanation is that there is an overhead to start searching results as soon as possible. However, it is the trade off to have better response time.

Chapter 5

Conclusions And Future Work

Keyword search in relational database attracts a lot of attentions in the database community over past decade. Most of fundamental work has been done. For example, an answer for a keyword search is no longer a table of records as in results for SQL queries. Instead, it is a graph of tuples that are connected by foreign key references. To find such an answer, the database is considered as a directed graph, in which nodes are tuples and edges are foreign key references. Since there are existing algorithms in graph theory, people proposed solutions based on graph algorithms to find answers for keyword queries. They are categorized as Graph-Based KWS-R systems [4, 20, 14, 9, 11, 22]. Some other people, on the other hands, proposed solutions based on the existing technologies of traditional databases, especially the schemas table that all databases have. They are categorized as Schema-Based KWS-R systems [1, 17, 15, 37, 26, 27, 39]. In both kinds of systems, however, if the system cannot find tuples that contain all the keywords and also are connected by foreign key references, then no result will be returned. We call it *empty result problem*. Empty result problem happens firstly when the database is not totally connected. In other words, the data

graph of the database consists of many disconnected small graph instead of a single large graph. Secondly, even if tuples that contain all keywords are in a single graph, it also happens when tuples are very far away so that the size of final answer would be too large to be meaningful to the users.

In this thesis, we proposed a solution to empty result problem by adding a new type of connection. Our solution use the similarity between two tuples as a second connection. In addition to the foreign key references, the similarity increases the possibility of connection of any two tuples in the database. The basic idea of our solution is that we first find all the graphs of tuples that contain only a portion of keywords in the query, called partial results. We then use similar tuples to expand partial results. Finally we find combinations of the expanded results that contain all the keywords. Our system is triggered whenever a keyword query generates empty results in the traditional KWS-R system.

To define the similarity of two tuples, we consider each tuple as a small document. We use TF-IDF weighing technology from Information Retrieval to convert each tuple into a term vector. Finally the *cosine* value of two term vectors is computed as the similarity of the two corresponding tuples. In order to quickly find similar tuples, we also build indexes that are modified from inverted indexes to boost searching time. We design two algorithms, ExpandTuple and ProgressiveExpandTuple to find similarity connection between partial results, and return those combinations of partial results that contain all keywords. To rank the results, we modified the IR-style ranking function in Schema-Based KWS-R [39].

We also conduct experiments to evaluate the effectiveness and the performance of our system. From the experiments results, we show that although our system is not guaranteed to return very high quality results for every query, it is capable of finding the most relevant answers for some queries and the average quality of top 10 answers is also good. Moreover, our system takes only 0.5 to 1.5 times extra time to search for results when the empty result problem

happens.

One possible future work is improving the performance of the algorithms. Another interesting topic is the definition of the similarity between tuples. The current definition is not applied to every database. Is there a more general definition or for a specific database, can we have a better definition of similarity? This could also be a future work. For databases that are updated frequently, incremental update to similarity indexes is also very important, because it is expensive to pre-compute indexes. Therefore, it is also a good topic to work on in the future.

Bibliography

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: a system for keyword-based search over relational databases. In *Proceedings of the 18th International Conference on Data Engineering*, pages 5 –16, 2002.

[2] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[3] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: authority-based keyword search in databases. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 564–575, 2004.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings of the 18th International Conference on Data Engineering*, pages 431 –440, 2002.

[5] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. VLDB Endowment, 2004.

[6] J. Coffman and A. C. Weaver. A framework for evaluating database keyword search strategies. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pages 729–738, New York, NY, USA, 2010. ACM.

[7] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 1998.

[8] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. In *Proceedings of the VLDB Endowment*, volume 1, pages 1189–1204, 2008.

[9] B. Ding, J. Xu Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 836–845, 2007.

[10] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 2007.

[11] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 927–940, 2008.

[12] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *Proceedings of the 12th International Conference on World Wide Web*, 2003.

[13] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1998.

[14] H. He, H. Wang, J. Yang, and P. Yu. Blinks: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 305–316, 2007.

[15] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 850–861, 2003.

[16] V. Hristidis, H. Hwang, and Y. Papakonstantinou. Authority-based keyword search in databases. *ACM Transactions on Database Systems*, 33(1):1–40, 2008.

[17] V. Hristidis and Y. Papakonstantinou. Discover: keyword search in relational databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 670–681, 2002.

[18] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*. North-Holland, 1992.

[19] M. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[20] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 505–516, 2005.

[21] W. Kießling and G. Köstler. Preference sql - design, implementation, experiences. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 990–1001. VLDB Endowment, 2002.

[22] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 173–182, 2006.

[23] G. Li, J. Feng, F. Lin, and L. Zhou. Progressive ranking for efficient keyword search over relational databases. In *Proceedings of the 25th British National Conference on Databases*, pages 193–197, 2008.

[24] G. Li, B. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured

data. In *Proceedings of the 2008 SIGMOD International Conference on Management of Data*, pages 903–914, 2008.

[25] G. Li, X. Zhou, J. Feng, and J. Wang. Progressive keyword search in relational databases. In *Proceedings of the 25th International Conference on Data Engineering*, pages 1183 –1186, 2009.

[26] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 563–574, 2006.

[27] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 115–126, 2007.

[28] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[29] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 605–616. ACM, 2007.

[30] A. Markowetz, Y. Yang, and D. Papadias. Keyword search over relational tables and streams. *ACM Transactions on Database Systems*, 34(3):1–51, 2009.

[31] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, 2009.

[32] A. Motro. Query generalization: a method for interpreting null answers. In *Proceedings from the first international workshop on Expert Database Systems*, pages 597–616, 1986.

[33] A. Motro. Seave: a mechanism for verifying user presuppositions in query systems. *ACM Transactions on Information Systems*, 4(4):312–330, 1986.

[34] L. Qin, J. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 681–694, 2009.

[35] L. Qin, J. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *Proceedings of the 25th International Conference on Data Engineering*, pages 724 –735, 2009.

[36] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[37] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 346 –355, 2007.

[38] W. Webber. Evaluating the effectiveness of keyword search. *IEEE Data Engineering Bulletin*, 33(1):55–60, March 2010.

[39] Y. Xu, Y. Ishikawa, and J. Guan. Effective top-k keyword search in relational databases considering query semantics. *APWeb/WAIM 2009 International Workshops*, pages 172–184, 2009.

Chapter 6

Appendix

6.1 Queries

Table 6.1 shows the 50 queries for IMDb dataset that are used in our experiments. Table 6.2 shows the 50 queries for Wikipedia dataset.

washington teaching ball	clint fighting waitress
john stomach	will jackie japan
ford empire	roberts people world
tom jobs	depp guitarist keith
go wind	space wars
africa lover	god rings
sound euphony	magic of oz
the note book	vietnam gump
precious bride	coppola father
atticus lawyer	jones professor
james bourne	rick paris
gunman kane	hannibal mills
bates robert bloch	vader samurai
wicked west magic	ratched joker
apparently my dear i don't give a damn	i'm going to make him an offer he can't reject
i'm going to give him an offer he can't reject	toto i've a feeling we're not in kansas any more
here is looking at you child	hamill vader
forrest 2004	henry fonda you my ours character
russell crowe bullfight	spiner star alien
hepburn boardway fiction	clouseau detective
name ryan rabbit	rocky arnold
name ryan red	rocky alois
name sky net back	ford lucas western
sean connery alexander	reeves brother manga
dean jones carlo	indiana jones 1989 robot wars

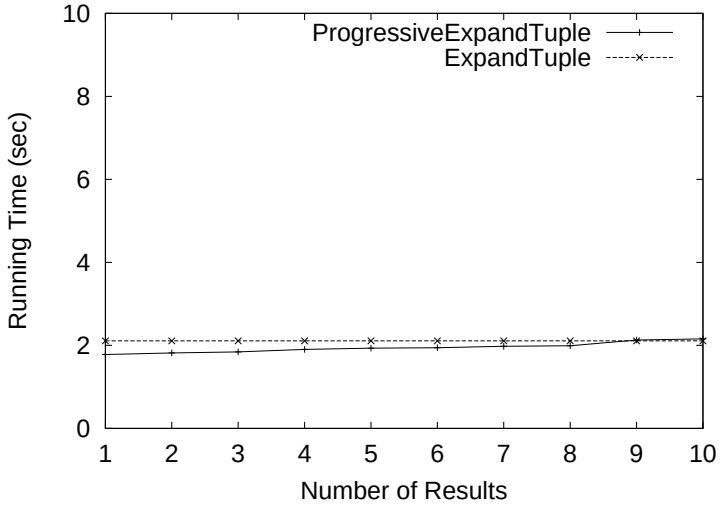
Table 6.1: Queries for the IMDb dataset

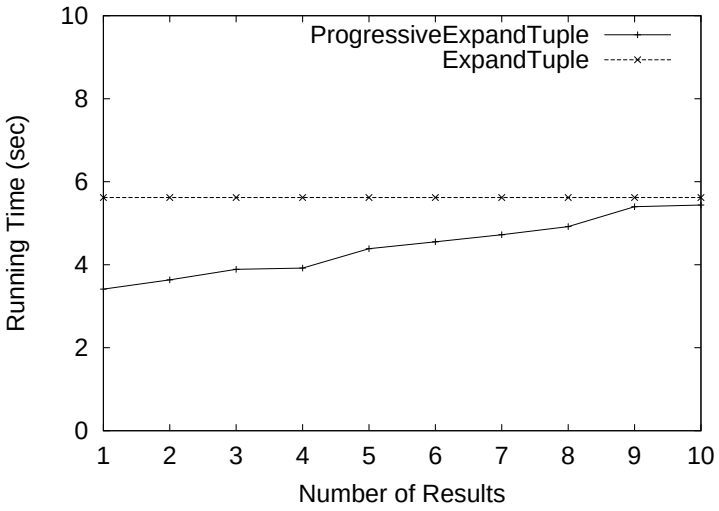
explorer probe	rocks rhapsody
1755 lisbon collapse building	nuba wood
entertain circus street	leap year explorer louis
scott north pole	monty flying circle
british east india fire mountain	globalization english
india antiquity plateau	worldwise english
bird white black chicken	yuhuangge chongqing china district beijing
kindle basin brazil	gas tungsten arc welding metallic shield
bezos basin brazil	gas tungsten arc welding metal shell
moorhen gallinula chik	england france hawfinch
hilda of whitby christmas saint	roman eifel civilization building
fsa finance services agent	armstrong china circle france
fall apart history 1989	breakdown of powers
hongkong landmark finance china	takashi tezuka fable
atlantic japan eastern battle	george edward fascistic
19 galveston	galveston typhoon
dover steep edge rock	somalia battle
william pitt junior	russia union casualties world war ii
dam lake patriot	nonrational number
mona lisa model	spanish swine
exxon valdez oil wasteweir	einstein special relation
pridefulness and prejudice author	nuclear numbers lanthanides
smallpox efficacious	web AlleborgoBot
mwuser weapon hiroshima	worldwise developing world bank
england rugby city	speak civilization

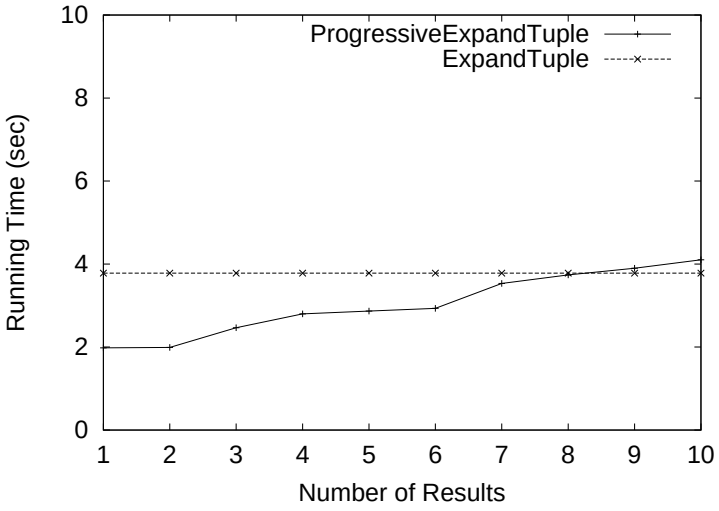
Table 6.2: Queries for the Wikipedia dataset

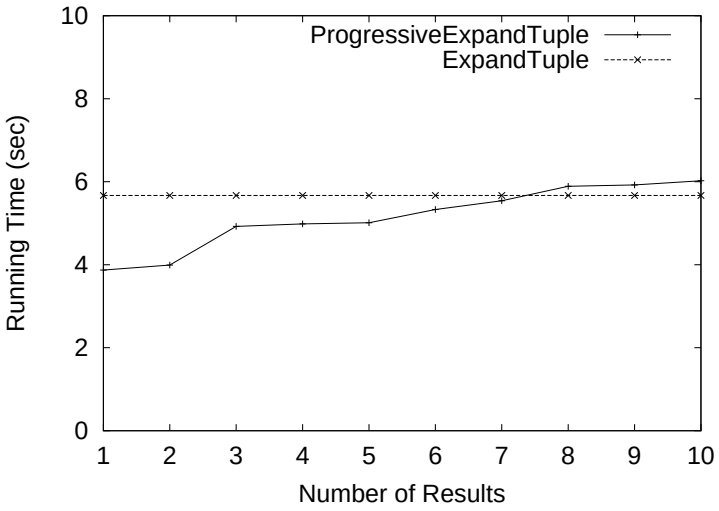
6.2 Comparison of ExpandTuple and ProgressiveExpandTuple

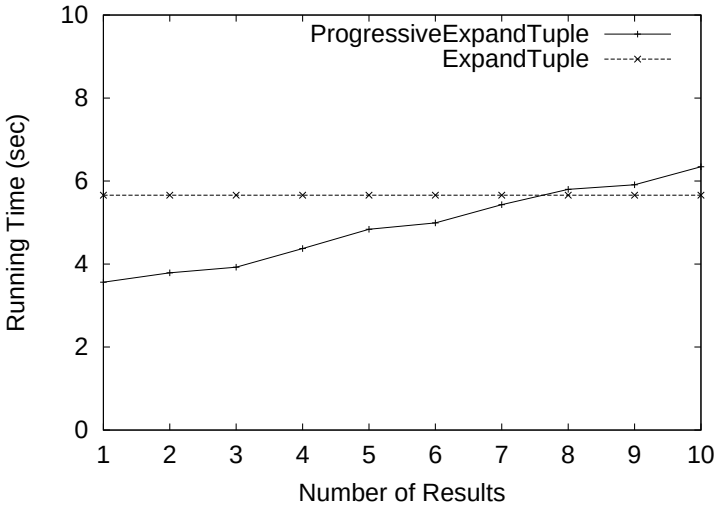
The following 10 figures shows that the comparison of ExpandTuple and ProgressiveExpandTuple for the 10 queries we use in our experiments. We can notice that in 9 of 10 queries, ProgressiveExpandTuple uses more time than ExpandTuple to return the first 10 results, although it returns the first 1 result much faster than ExpandTuple does.

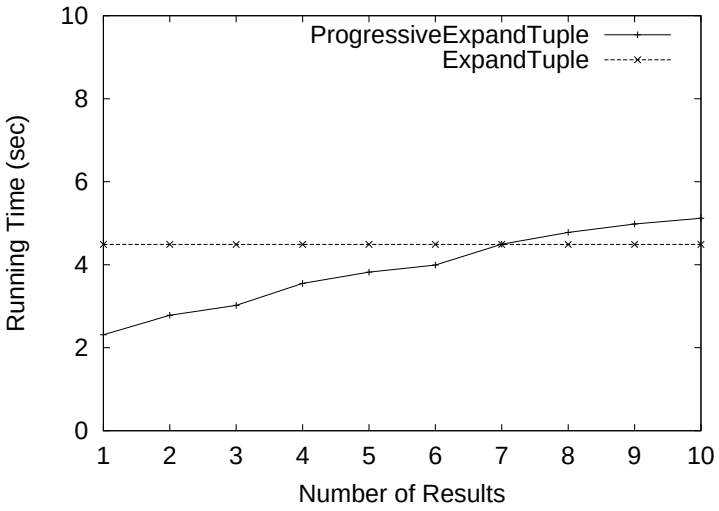


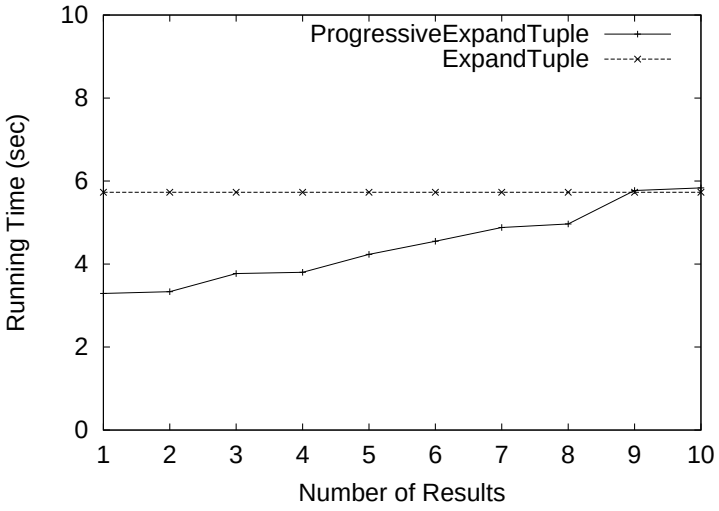

ProgressiveExpandTuple
ExpandTuple
Running Time (sec)
Number of Results
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10

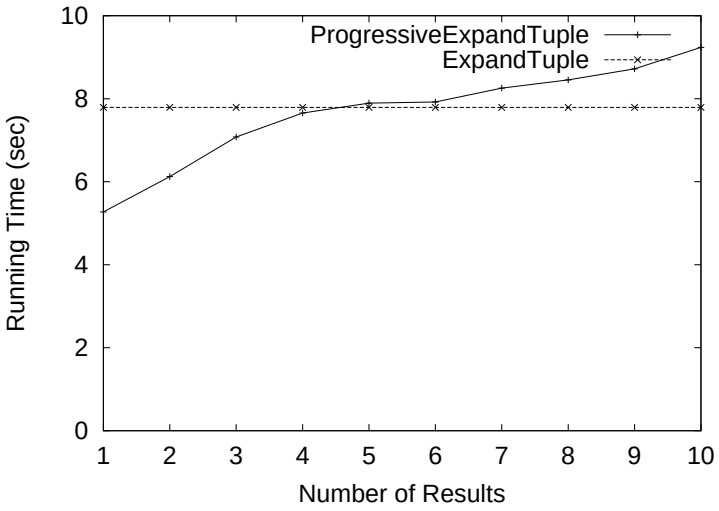

ProgressiveExpandTuple
ExpandTuple
Running Time (sec)
Number of Results
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10

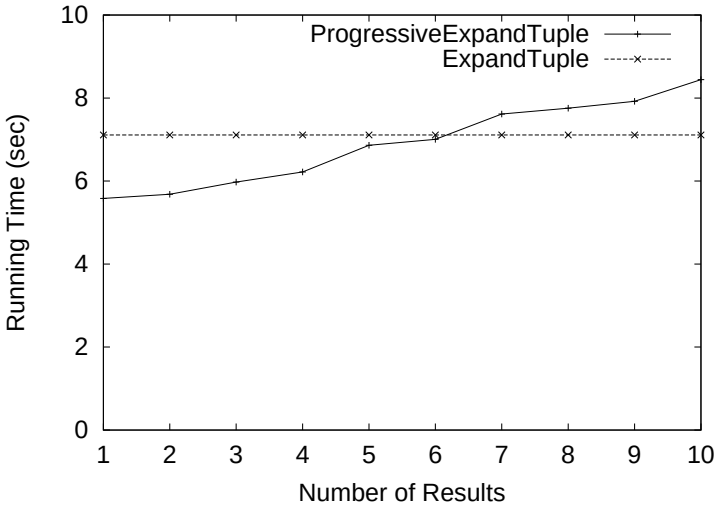

ProgressiveExpandTuple
ExpandTuple
Running Time (sec)
Number of Results
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10

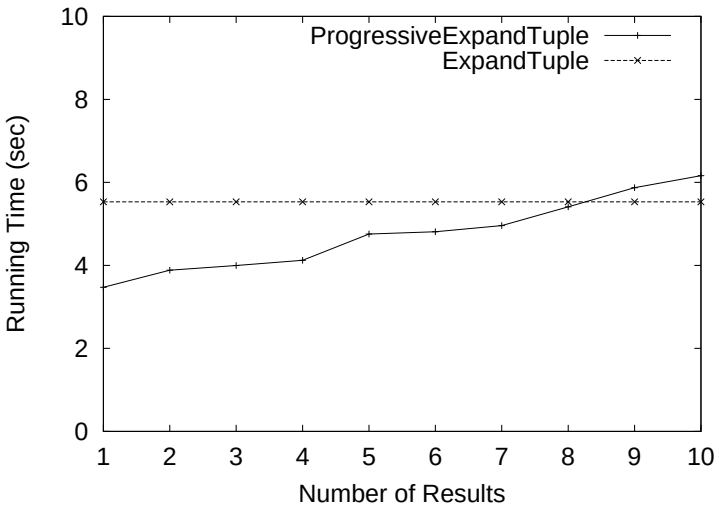

ProgressiveExpandTuple
ExpandTuple
Running Time (sec)
Number of Results
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10


ProgressiveExpandTuple
ExpandTuple
Running Time (sec)
Number of Results
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10


ProgressiveExpandTuple
ExpandTuple
Running Time (sec)
Number of Results
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10


ProgressiveExpandTuple
ExpandTuple
Running Time (sec)
Number of Results
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10


ProgressiveExpandTuple
ExpandTuple
Running Time (sec)
Number of Results
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10


ProgressiveExpandTuple
ExpandTuple
Running Time (sec)
Number of Results
0
2
4
6
8
10
1
2
3
4
5
6
7
8
9
10