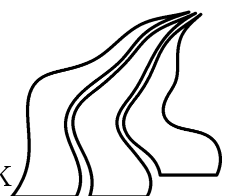
AALBORG UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
FREDRIK BAJERS VEJ 7 - DK 9220 AALBORG - DENMARK



Title:

Spiderlink: a Keyword Search Algorithm

Project period:
2003.02.01 – 2003.06.06

Members:
Linas Būtėnas

Advisor:
Michael Böhlen

Abstract:
The increasing need for a keyword-based search systems on relational databases motivated us to develop the SpiderLink search engine. It uses a k-tree data structure to find the connections between given keywords. As a result SpiderLink returns the sequences of tuples relating the tuples where keywords were found.
In paper we first define a *k-tree* data structure. It has several important properties: i) it works on hierarchical, parallel and single relationships in a database; ii) it is minimal; iii) it is finite; iv) it can be implemented as a hash-table allowing to use it most efficiently.
Later we present the SpiderLink search algorithm and diagram in the example how it actually works. We have implemented SpiderLink and it is a fully developed keyword search engine. The tests done on the databases support all our theoretical assumptions.

Article: 1

Pages: 23

Copies: 6

2003

Spiderlink: a Keyword Search Algorithm

Linas Būtėnas
linas@cs.auc.dk
Department of Computer Science, Aalborg University,
Frederik Bajers Vej 7E, 9220 Aalborg Øst, Denmark

June 6, 2003

Abstract

The increasing need for a keyword-based search systems on relational databases motivated us to develop the SpiderLink search engine. It uses a k-tree data structure to find the connections between given keywords. As a result SpiderLink returns the sequences of tuples relating the tuples where keywords were found.

In paper we first define a *k-tree* data structure. It has several important properties: i) it works on hierarchical, parallel and single relationships in a database; ii) it is minimal; iii) it is finite; iv) it can be implemented as a hash-table allowing to use it most efficiently.

Later we present the SpiderLink search algorithm and diagram in the example how it actually works. We have implemented SpiderLink and it is a fully developed keyword search engine. The tests done on the databases support all our theoretical assumptions.

1 Introduction

The popularity of keyword search systems shows that keyword searches are the most convenient way of retrieving the information from huge data sources. It is simple and precise way to ask and get the information user wants. The World Wide Web already has a number of well developed keyword search systems, like Google [5], Altavista and others. But there is an increasing need for keyword-based search engines for relational databases.

In this paper we propose the SpiderLink – a keyword search engine on relational databases. It uses a k-tree data structure to find the connections between given keywords. As a result user gets the sequences of tuples relating the tuples where given keywords were found.

Our main goal is to present the k-tree data structure. It is the core of our SpiderLink algorithm. A k-tree has several very important features:

- it can be used for keyword searching on various databases having single, hierarchical and parallel relationships.
- it is minimal and guarantees that it will take the smallest amount of space to represent n tuples and n-1 relationships between them.
- it is finite. This feature gives a possibility for algorithm to build a k-tree till it is possible to do that, and we are sure there is an end for building process.
- an implementation of it can be done in various ways. We propose to implement it using a hash-table. Hash-table gives a constant time performance while searching for a node in a k-tree.

Our second goal is to present the keyword search algorithm SpiderLink. In paper first we describe the algorithm and after give an example showing step by step how the algorithm works. We have also implemented SpiderLink and in one of the sections we present the architecture of this implementation. An implemented SpiderLink engine works as a fully functioning keyword search system for relational databases. In addition to the properties possessed by k-trees it has some very useful features for a user. SpiderLink finds the relationships between keywords not only in the case then keywords are located in the tuples, but also in the case, where keywords are located in the names of relations or attributes. It finds exact and not exact keyword matches, and it does not require from user any knowledge of database schema or database querying language (such as SQL).

In the article we discuss how to find the keywords in the database, the data structures used for constructing and checking the relationships between these keywords, and the ranking and pruning of results.

The paper has the following structure. In the Section 2 we discuss the related work and in Section 3 we give a motivation. Preliminary definitions are given in Section 4. The data structure of a k-tree is discussed in Section 5. In Section 6 we describe the SpiderLink algorithm. The following Section 7 gives an example how the algorithm works. In Section 8 we discuss the architecture of SpiderLink implementation.

2 Related Work

In the recent years there has been a number of papers published in the area of keyword searching in relational databases. All of them attempt to give a simple keyword search interface, short searching time and wide range of results. To satisfy all of these needs is a challenging task. Usually one of them is solved in a trade-off of the other. Most of the papers takes a part of all the problems and tries to solve them.

Most of the relational database management systems like DB2 [10], Microsoft SQL Server [8], MySQL, Oracle [6] and PostgreSQL have extensions, providing text

search engines. However, these search engines can perform a text search only on single columns. That is the reason why there is a need for keyword search engines, capable to find relationships between keywords.

The paper [4] describing AQUA/Jungle database search engine is proposing an idea of advertising databases via Internet. This prototype and our SpiderLink algorithm works towards the same general idea: they give a user an easy access to the information stored in databases. But the chosen searching strategies are different. Jungle first indexes meta-data and later this information is used for optimization of SQL queries in AQUA. The purpose of SpiderLink algorithm is to work directly with database without indexing any information and to issue SQL queries on single relation only.

There are several Internet search engines like Quigo or iBoogie that have implemented the Deep Web [2] search techniques. Such type search engines are able to find information in e-commerce or news web sites databases. Dynamically generated World Wide Web pages are not always reachable for standard WWW keyword search engines.

The most related work to our algorithm is described in the following four papers [1, 3, 7, 9]. All of them aims to provide a simple interface for a database access, do not require any knowledge of database schema and database querying language. But the searching strategies differs.

Papers [7] and [3] treats a database like a huge graph, where tuples are nodes and relationships are edges. Viewing from this point our work is quite relative to what they did. We build a data structure where tuples are also treated as nodes and relationships are references between them. The difference is that paper [7] describes a system for keyword searches in graph-structured databases. It focuses on ranking the results, based on shortest path computations. The BANKS system [3] loads the whole relational database as a weighted graph. After that it uses Steiner trees [1] to find the closest connections between tuples where keywords were found.

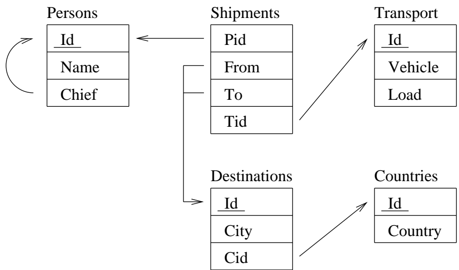
The DISCOVER [9] and DBXplorer [1] keyword search algorithms chooses another approach. They explore the properties of the database schema. First it finds keywords in the database and later chooses one which becomes a starting point for building the candidate paths between all keywords. DISCOVER uses schema graph to generate *minimum joining networks of tuples*, which can be joined together into a single network containing all of the keywords. It uses an execution plan to optimize and reuse issued SQL queries.

The DBXplorer [1] locally stores database schema and index of keywords. It finds a set of relations where given keywords are located, using the keyword index. Then using the undirected schema graph it builds a set of subgraphs representing a joining of relations. Each candidate subgraph is issued as an SQL statement to check if it contains all the keywords.

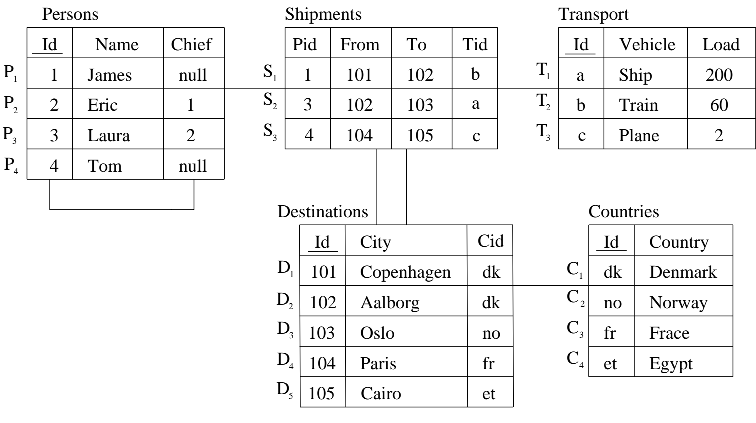
[1] Steiner tree – a minimum-weight tree connecting set of vertices in a weighted graph.

3 Motivation

In this section we introduce a sample database and a motivating example for our keyword search algorithm SpiderLink. Consider the database schema in Figure 1(a) and the sample instance in Figure 1(b). The database shows information about persons transporting goods between cities. The example contains a relation having a relationship to itself ("Persons"), two parallel relationships from relation "Destinations" to relation "Shipments", a ternary relationship and several single relationships. Such structure of the example was chosen to represent the possibilities of our search engine and to point out the most common situations in the real world databases.



(a) A schema of database. The underlined attributes are primary keys. Arrows go from foreign keys to primary keys.



Persons

	Id	Name	Chief
P_1	1	James	null
P_2	2	Eric	1
P_3	3	Laura	2
P_4	4	Tom	null

Shipments

	Pid	From	To	Tid
S_1	1	101	102	b
S_2	3	102	103	a
S_3	4	104	105	c

Transport

	Id	Vehicle	Load
T_1	a	Ship	200
T_2	b	Train	60
T_3	c	Plane	2

Destinations

	Id	City	Cid
D_1	101	Copenhagen	dk
D_2	102	Aalborg	dk
D_3	103	Oslo	no
D_4	104	Paris	fr
D_5	105	Cairo	et

Countries

	Id	Country
C_1	dk	Denmark
C_2	no	Norway
C_3	fr	Frace
C_4	et	Egypt

(b) Database instance

Figure 1: Sample database. It has hierarchical relationship in relation "Persons", parallel relationship between relation "Shipments" and relation "Destinations", and a ternary relationship between 4 relations.

Assume a user would like to find if James could transport by ship a cargo between Aalborg and Oslo. The appropriate keyword query for SpiderLink is "James ship

Aalborg Oslo". Now the search engine should find the tuples where these four keywords are located. From the example in Figure 1(b) we see that tuple P_1 contains keyword "James", tuple T_1 contains keyword "Ship", tuple D_2 contains "Aalborg" and finally tuple D_3 contains "Oslo". The second step for SpiderLink is to find the sets of tuples that connect P_1, T_1, D_2 and D_3. The answer from a search engine would show they are connected, but not in a straight forward way. James himself does not transport anything between Aalborg and Oslo, but his subordinate Eric does.

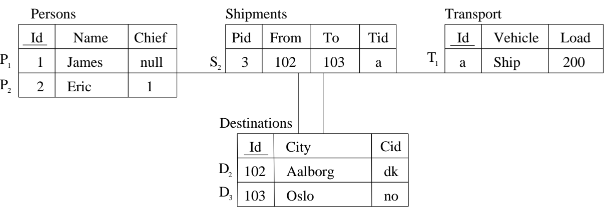


Figure 2: A graphical representation of search results

One of the answers that SpiderLink outputs is these two sets of related tuples: $\{P_1, P_2, S_2, T_1\}$ and $\{D_1, S_2, D_3\}$. They represent relationships between four tuples where keywords were found. A textual representation of results is not so self explicit as the graphical one in Figure 2, but this is just a matter of way in which we represent the result. Our main concern for SpiderLink algorithm is to return short and non duplicate results. We don't confuse user by giving the same results many times represented by different sets of tuples. Consider this example: $\{P_1, P_2, S_2, D_2\}$, $\{P_1, P_2, S_2, D_3\}$, $\{T_1, S_2, D_3\}$. The actual result is completely the same as the previous one, and even the graphical representation is the same like in Figure 2. If a user gets many results like these it will take time to figure out which results are new and which ones was already presented before.

In the case we have presented above, a set of tuples are related to each other through the primary-foreign key relationships. A set of related tuples is an answer if it contains all the keywords in the tuples or the keywords match corresponding attribute or relation names. We showed one example in order to illustrate what is the keyword search query input, the result, and the meaning of the result.

4 Preliminary definitions

A *Database* $D = (\mathbb{R}, \mathbb{RL})$ is a pair of sets: set of relations $\mathbb{R}$ and set of relationships $\mathbb{RL}$. Each relation $R = (S^R, \{R_1, \ldots, R_n\})$ is a pair consisting of schema S^R and a set of tuples $\{R_1, \ldots, R_n\}$ over the schema S^R. Schema $S^R = \{A_1, \ldots, A_n\}$ is a set of attributes. Tuple $R_i = \{v_1, \ldots, v_n\}$ of the relation R is a mapping from the attributes of the schema S^R to the set of values $\{v_1, \ldots, v_n\}$. A number of tuples in

a database is finite as there is a finite number of relations and every relation has a finite number of tuples.

We assign a unique identifier for each tuple in the database. We use R_i as this identifier. It shows exactly to the tuple i in the relation R. No other tuple has the same identifier in the same database. In Figure 1(b) there is a database example where you can see these identifiers written at the side of each tuple. The SpiderLink algorithm uses these identifiers to track relationships between locations of keywords in a database. In terms of memory consumption it is much cheaper to keep only identifiers of tuples instead of whole tuples.

A relationship $RL = \{(R.A_1, R'.A'_1), \ldots, (R.A_n, R'.A'_n)\}$ is a set of pairs of elements, representing a connection between two relations. We will write that $R, R' \in RL$ if $RL = \{(R.A_1, R'.A'_1), \ldots, (R.A_n, R'.A'_n)\}$ and say that RL connects R and R'. Attributes A_i and A'_i sitting in one element will be called *(a pair of) related attributes*. We use a set of elements instead of one element in relationship, because a relationship in a database could involve more than one attribute from each relation. It could be understood like a set of attributes $\{A_1, \ldots, A_n\}$ is a primary key in relation R and set $\{A'_1, \ldots, A'_n\}$ forms a foreign key in relation R' referencing primary key in R.

Example 4.1 Here we start a series of examples using database instance in Figure 1(b). With a help of it we explain and diagram visually the search process and terms we define. According to our database description the database instance is defined like that:

$Database = (\{P, S, T, D, C\}, \{RL^{PS}, RL^{ST}, RL_1^{SD}, RL_2^{SD}, RL^{DC}\})$ [23]

Relations:	Relationships:
$P = (\{Id, Name, Chief\}, \{P_1, P_2, P_3, P_4\})$	$RL^{PP} = \{P.Id, P.Chief\}$
$S = (\{Pid, From, To, Tid\}, \{S_1, S_2, S_3\})$	$RL^{PS} = \{(P.Id, S.Pid)\}$
$T = (\{Id, Vehicle, Load\}, \{T_1, T_2, T_3\})$	$RL^{ST} = \{(S.Tid, T.Id)\}$
$D = (\{Id, City, Cid\}, \{D_1, D_2, D_3, D_4, D_5\})$	$RL_1^{SD} = \{(S.From, D.Id)\}$
$C = (\{Id, Country\}, \{C_1, C_2, C_3, C_4\})$	$RL_2^{SD} = \{(S.To, D.Id)\}$
	$RL^{DC} = \{(D.Cid, C.Id)\}$

Definition 1 (Adjacent tuples). Tuple R_i is *adjacent* to tuple R'_j if there is a relationship connecting relations R and R' and for each pair of related attributes in that relationship values are the same in both tuples.

Tuple $R_i = \{v_1, \ldots, v_n\}$ is *adjacent* to tuple $R'_j = \{v'_1, \ldots, v'_n\}$ if:

1) exists a relationship RL, such that $R, R' \in RL$;

2) for each element $(R.A_i, R'.A'_j)$ from the relationship RL the condition is satisfied: $v_i = v'_i$, where v_i is a value mapped from attribute A_i in the relation R and v'_i is a value mapped from attribute A'_i in the relation R'.

[2]We use an abbreviations for the relation names, writing only the first letters.

[3]An index shows the names of connected relations by this relationship.

Definition 2 (Related tuples, Path). Tuple R_1 is a *related tuple* to tuple R_n if there exists a sequence of tuples $(R_1, \ldots, R_n)$ where next to each other standing elements R_i, R_{i+1} are adjacent tuples. This sequence of tuples will be called a *path* between R_1 and R_n.

Example 4.2 Lets take a tuple P_1 from a relation "Persons" and analyze it. Following Definition 1 we find out that tuples P_2 and S_1 are adjacent to it. There are quite a lot of related tuples to tuple P_1, so we take only two as examples: P_3 and T_2. The path between P_1 and P_3 is (P_1, P_2, P_3) as well as $(P_1, S_1, D_2, S_2, P_3)$ and many others. As you can see, there are many possible variations of putting down a path between two tuples.

5 Data Structures

x In this section we describe a data model of our search engine. First we define a data structure of a k-tree and explain the rationale of the design. We also describe the notion of k-path and mk-path as they are essential elements of a result that user gets.

User starts the search by giving a stream of keywords. This stream is parsed to terms and every single term becomes a *keyword* KW. A set of all keywords we mark as $\mathbb{KW}$.

Example 5.1 Lets assume we would like to find out if James transfers goods by train in Denmark. The keyword input stream looks like "James train Denmark". This stream is parsed and we get three keywords: KW_1="James", KW_2="train", KW_3="Denmark". A set of all keywords $\mathbb{KW} = \{KW_1, KW_2, KW_3\}$.

After we have parsed the keywords, a database is scanned searching for the locations of these keywords. A *location of the keyword* is a tuple R_i containing that keyword. If a keyword is founded in the name of relation or in the name of attribute, then we say that keyword is founded in every tuple of that relation and every tuple becomes a location of that keyword. We define a function $kw(R_i)$: if R_i is a location of keyword KW than $kw(R_i) = KW$.

Example 5.2 In our database example in Figure 1(b) keyword "James" is found in tuple P_1. The location of the keyword "train" is the tuple T_2, and location of the keyword "Denmark" is the tuple C_1. In Figure 3 you can see all three tuples grouped by keywords.

5.1 The K-tree

The k-tree data structure should satisfy several requirements:

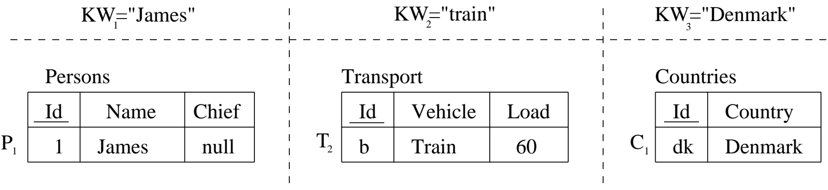


Figure 3: Keyword locations

1. contains a location of a keyword as a root node;
2. represents relationships between the location of the keyword (root) and other tuples in a database (other nodes);
3. supports possibility to reconstruct a path between a leave an a root of the tree;
4. allows to have an unlimited number of children for each node;
5. supports different types of nodes in order to minimize the total amount of nodes in the tree;
6. nodes are sorted in order to give a fast access to required type;
7. don't run into cycles then there are cyclic relationships in a database;
8. keeps a minimal number of nodes without affecting the requirements pointed above.

A k-tree T is a set of nodes. Every node N consists of two elements: a tuple identifier R_i and a reference to a parent node N_{Parent}: $N = (R_i, N_{Parent})$. As illustrated in Figure 4, the pointers in a k-tree point from children to their parents. The reason is that nodes can have an unlimited number of children, but only one parent. We define two auxiliary functions: *tid(N)* and *parent(N)*. *tid(N)* returns the tuple identifier of node N: $tid(N) = R_i$. *parent(N)* returns the parent of node N: $parent(N) = N_{Parent}$.

Every k-tree has a root node as a starting point. We will refer to N_{Root} as to the root node of a k-tree. Root has no parent, so $parent(N_{Root}) = null$. The name of tuple in a root node is special - this tuple is a location for some keyword. To denote, that a root node of the k-tree T has a tuple as a location for a keyword KW, we will write $rootkw(T) = KW$.

Definition 3 (K-tree) A k-tree T is a union of three sets: a set of active nodes $\mathbb{N}_A$, a set of idle nodes $\mathbb{N}_I$ and a set of parent nodes $\mathbb{N}_P$. All together these sets form one k-tree $T = \mathbb{N}_A \cup \mathbb{N}_I \cup \mathbb{N}_P$. Every node in this tree fulfill three requirements:

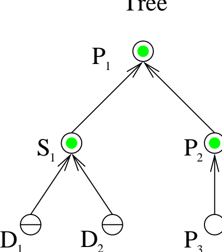


Figure 4: A fragment of a k-tree

1. every node is unique, e.i. $\forall N, N'\ tid(N) \neq tid(N')$, $N, N' \in T$.

2. every node (except root) has a parent in the same tree and their tuples are adjacent: $\forall N\ \exists N_{Parent}$ that $N_{Parent} = parent(N) \wedge$ tuple $tid(N)$ is adjacent to tuple $tid(N_{Parent})$, $N \in T \backslash N_{Root}$, $N_{Parent} \in \mathbb{N}_P$. $parent(N_{Root}) = null$.

3. from every node (except root) it is possible to reach the root node by following references to parents: $\forall N\ \exists$ sequence $(N, \ldots, N_{Root})$ where for every next to each other standing nodes N', N'', N' is a child of N''.

From the first requirement of Definition 3 follows a property that $(\mathbb{N}_A \cap \mathbb{N}_I) = (\mathbb{N}_A \cap \mathbb{N}_P) = (\mathbb{N}_I \cap \mathbb{N}_P) = \emptyset$. This means that every set does not intersect with the other sets in a k-tree. If a node belongs to one set, we are sure that it is not contained in the other ones.

Example 5.3 In this example we describe a tree fragment shown in Figure 4.
$Tree = \{P_3\}_A \cup \{D_1, D_2\}_I \cup \{P_1, P_2, S_1\}_P$.

The biggest amount of work is done with leaves of a tree. The other nodes are just used to restore a path between a leaf node and the root of the tree. The leaves are also used for different purposes. To be able to distinguish quickly between various types of nodes and to give fast access to the needed type we distribute them between three different types, where two of them cover leaves and the third is for parents. There is a short description of all types of nodes in Table 1. In Figure 4 you can see a fragment of a tree having nodes of all types. All of them are drawn according to their type.

In Figure 5 there is shown a life cycle of a node. In this picture there are presented different states of a node during the search process. There are also shown two processes from search algorithm which change the type of node. At this point it is important to understand only that after the leave takes part in these processes, its type is changed and there is no way it could be restored to the previous one.

Node type	Set	Image	Used for:
Parent	$\mathbb{N}_P$	◉	rebuilding a path between leaves and root
Leaf active	$\mathbb{N}_A$	○	making connections between trees and tree expansion
Leaf idle	$\mathbb{N}_I$	⊖	making connections between trees

Table 1: Types of nodes

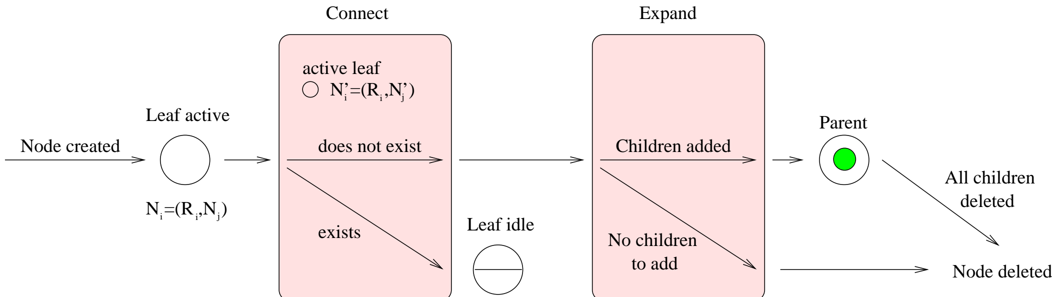


Figure 5: A life cycle of a node

Figure 6 diagram the usage of different types of nodes. Here the main point is to show why we need three types of nodes. All of them takes part in different processes.

We can clearly distinguish between nodes and use the appropriate set when it is needed. But still we will often encounter a situation where we have to find if a node is contained in some set. In this case we would like to do better than a linear scan. Hence, we store each set of nodes in a hash table. A key in this hash table is a name of a tuple and a value is a reference to the parent. Hash tables helps to search and retrieve a node in a constant time independently from the amount of nodes in a tree. This will let us minimize the overall time spent for searching. Another issue is to minimize a number of nodes stored in a tree. This would allow us to save the space and time (used for rehashing the tables).

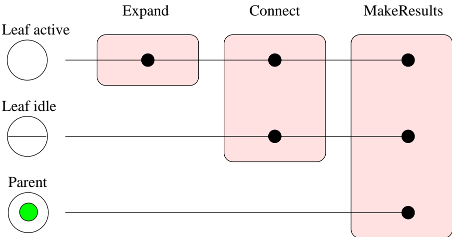


Figure 6: Node usage

Lemma 1 A k-tree with n nodes represents n tuples and $n-1$ relationships between these tuples.

Proof: According to requirement 1 in Definition 3 every node in a k-tree has different tuple. Thus, n nodes will have n different tuples. Also, according to requirement 2 in Definition 3 every node represents a relationship between tuple it has and the tuple of the parent node. In total there will be $n-1$ relationships as there are n nodes and root node doesn't have a parent. □

We have defined the structure of the k-tree, but from the definition itself it is not clear if k-tree is a real tree. We still need to prove that k-tree has no cycles.

Theorem 1 (K-tree is acyclic) The K-tree data structure contains no cycles.

Proof: We prove this theorem by contradiction. We will try to construct a cycle of nodes from the tree T. Cycle is a sequence of nodes $(N_1, \ldots, N_n)$, that:

1) N_{i+1} is a parent node of N_i;

2) N_1 is a parent node of N_n.

If we succeed to show that it is impossible to construct such cycle - the theorem will be proved.

Assume we have a tree T. N_{Root} is a root node of the tree T. Lets also assume that the tree T is large enough to make a sequence $(N_1, \ldots, N_n)$, where N_{i+1} is a parent node of N_i. Now we have a sequence complying with the first requirement for the cyclic sequence.

According to the definition of the k-tree (Definition 3) we can make a sequence of nodes between any node and a root node. Thus, we can make a sequence $(N_n, N_{n+1}, \ldots, N_{Root})$. Since sequence $(N_n, N_{n+1}, \ldots, N_{Root})$ is based on the same rules as the sequence $(N_1, \ldots, N_n)$ we can join them together. The result will be a sequence $(N_1, \ldots, N_n, N_{n+1}, \ldots, N_{Root})$. In this case we see that the parent of N_n is the node N_{n+1}. According to Definition 3 all nodes in the tree T are unique, so $N_{n+1} \neq N_1$ and node N have only one parent. Therefore N_1 is not a parent for N_n.

This proof shows that any time we make a sequence $(N_1, \ldots, N_n)$, we can extend it till the root node and every time we will get that N_1 is not a parent for N_n. Thus we have to conclude that it is impossible to make a cyclic sequence of nodes $(N_1, \ldots, N_n)$ from the k-tree T. Therefore k-tree T is acyclic. □

This theorem gives a nice feature that whenever we take a node and follow the references of parents, we will always end at the root node and will never run into infinite cycle. This property allows us to build a simple procedure in the algorithm for rebuilding a path from leave node to root node. A procedure is very simple as we have to follow only the references to parents. Theorem 1 ensures that we always one result for one leave node and that this process will never run into cycle.

Theorem 2 (Minimality of k-tree) K-Tree has the smallest number of nodes (number n) representing n tuples and $n-1$ relationships.

Proof: We prove this theorem by contradiction. Assume we have a k-tree T with n nodes in it. According to Lemma 1 this k-tree represents n tuples and $n-1$ relationships between these tuples.

Lets remove one node from the k-tree T and name a new k-tree T'. T' has $n-1$ nodes. In this case Lemma 1 says that the k-tree T' represents $n-1$ tuples and $n-2$ relationships between these tuples.

We have to conclude that reducing the number of nodes in a k-tree will result in reduced number of tuples and relationships. We wanted to prove that n tuples and $n-1$ relationships can be represented by the smaller number of nodes than n. The proof shows that this is impossible. The theorem is correct. □

Theorem 2 says that we will always have a minimal number of nodes in a k-tree. This feature is valuable as we can be sure that an overall number of nodes in all of the k-trees will also be minimal. So the space is used in most efficient way by the SpiderLink algorithm.

Theorem 3 (K-Tree is finite) K-Tree has a finite number of nodes.
Proof: According to Definition 3 all nodes in a k-tree have different tuple names. This means that the maximum number of nodes in a k-tree is limited by the number of tuples in a database. A database has a finite number of tuples[4], hence the number of nodes in a k-tree is also finite. □

Theorem 3 gives us another nice property: we can build a tree until there is impossible to continue building process. A tree has a finite number of nodes, so the number of tree building steps are also finite. We will use this property to define the "Stop Criteria" in our algorithm.

5.2 K-path and MK-path

In previous part we presented the a k-tree. But a k-tree itself represents only relationships between one location of one keyword and a stack of nodes. As the result we have to give to user a data structure containing one or several sequences of tuples showing a connection between locations of all keywords. In this part of paper we define two data structures: k-path and mk-path. A k-path shows a relationship between two locations of two different keywords. A mk-path shows relationships between k number of locations of different keywords, where k is the number of all keywords.

Definition 4 (K-path). A path $P = (R_1, \ldots, R_n)$ between two different keywords KW and KW' is called *k-path*, if:
1) $kw(R_1) = KW$;

[4]This feature of database is described in Section 4

2) $kw(R_n) = KW'$;
3) $\forall R_i, R_j \in P \quad R_i \neq R_j$

Let $connects(P)$ return the two keywords connected by k-path P: $connects(P) = (KW, KW')$ if $P = (R_1, \dots, R_n)$ and $kw(R_1) = KW$ and $kw(R_n) = KW'$. Such strict definition is needed, because function $connects(P)$ outputs only two keywords even though there is a tuple R_i in the path P such that $kw(R_i) = KW''$.

From Definition 4 follows a property that if $P_1 = (R_1, \dots, R_i)$ and $P_2 = (R_i, \dots, R_n)$ are k-paths then a sequence (P_1, P_2) results in a new k-path $P = (R_1, \dots, R_i, \dots, R_n)$. We also have to mention, that if $connects(P_1) = (KW, KW')$ and $connects(P_2) = (KW', KW'')$ then $connects(P) = (KW, KW'')$.

Definition 5 (MK-path) Let $\mathbb{KP} = \{KP_1, \dots, KP_n\}$ be a set of k-paths connecting all keywords from $\mathbb{KW}$. This set we will call a *mk-path*. This set fulfills two requirements:
1) $n = size(\mathbb{KW}) - 1$;
2) $\forall KW, KW' \in \mathbb{KW}$ $\exists$ a subset $\{KP_i, \dots, KP_j\} \subseteq \mathbb{KP}$, such that a sequence $(KP_i, \dots, KP_j)$ of this subset is a k-path connecting KW and KW';

As a result user gets a number of mk-paths representing the relationships between all of the requested keywords.

6 Algorithm

6.1 Structure of the algorithm

Here is the section where we go through entire search process of the SpiderLink algorithm. Our algorithm consists of four basic steps:

1. **Initialization** - finds locations for the given keywords. Makes initial k-trees.
2. **Connect procedure** - makes k-paths using k-trees.
3. **MakeResults procedure** - makes new mk-paths if possible. Outputs them to user.
4. **Expand procedure** - expands k-trees by adding children to their leaves.

The last tree steps are performed iteratively until at least one condition in "Stop Criteria" list is satisfied. In the following part of this section we describe each of these parts in detail. First we present the general structure of SpiderLink algorithm.

Purpose: to find all connections between given keywords.
Input: a set of keywords $\mathbb{KW}$ and a database D. The input set of keywords is treated as a conjunctive query, so every answer should include all keywords.

Output: set of mk-paths.

```
SpiderLink(𝕂𝕎,D)
1. let 𝕋 be an empty set of k-trees
2. let ℙ be an empty set of k-paths
3. for each keyword KW ∈ 𝕂𝕎:
4.     for each relation R ∈ D:
5.         for each tuple {R_i|kw(R_i) = KW}:
6.             make a k-tree T with a root node N_Root = (R_i, null)
7.             add T to 𝕋
8.         end for
9.     end for
10. end for
11. if ∀KW ∈ 𝕂𝕎 ∃T that rootkw(T) = KW than
12.     while "Stop Criteria" is not satisfied:
13.         for each k-tree T ∈ 𝕋:
14.             ℙ_new = Connect(T, 𝕋)
15.             ℙ = MakeResults(ℙ_new, ℙ)
16.             T = Expand(T, D)
17.         end for
18.     end while
19. end if
```

An input for SpiderLink is a set of keywords $\mathbb{KW}$ and a database D. First of all algorithm finds tuples containing keywords from $\mathbb{KW}$. Then for each tuple R_i (location of a keyword) we make a new k-tree T with a root $N_{Root} = (R_i, null)$. After the database is scanned searching for all keywords, we have a set $\mathbb{T}$ of initial k-trees. These k-trees are our starting points for the iterative steps.

If we have found at least one location for each of the keywords from $\mathbb{KW}$, we start the iterations. In other way, if one of the keywords is not found, there is no reason to make further steps. A set of keywords $\mathbb{KW}$ is a conjunctive keyword query. This means that all of the keywords should be contained in the result. Thus, if one of the keywords is not found in a database at all – we stop the search process.

After the "Initialization" part (lines 3-10) follows three important steps and checking one condition. The "Stop Criteria" is described after the definitions of steps "Connect", "MakeResults" and "Expand". At that time it will be much easier to understand why such conditions define "Stop Criteria".

The following process is repeated until the "Stop Criteria" is satisfied. For every iteration we take a k-tree T from the set $\mathbb{T}$ and perform "Connect", "MakeResults" and "Expand" steps with it. Usually the number of iterations performed is greater than the number of k-trees in the set $\mathbb{T}$. Thus, every time we process the last k-tree T_n from the set $\mathbb{T}$, the next time we start again from the first k-tree T_1.

6.2 Connect

The first procedure is "Connect". It is a part of algorithm where k-paths are produced between locations of different keywords.

Purpose: to produce all possible k-paths.

Input: a tree T, a set of all trees $\mathbb{T}$.
Output: a set of k-paths $\mathbb{P}$.

```
Connect(T,𝕋)
1. let ℙ be an empty set of k-paths
2. for each leaf node {N|N ∈ ℕ_A ∪ ℕ_I , T = (ℕ_A ∪ ℕ_I ∪ ℕ_P)}:
3.     for each k-tree {T'|rootkw(T) ≠ rootkw(T'), T' ∈ 𝕋, T' = (ℕ'_A ∪ ℕ'_I ∪ ℕ'_P)}:
4.         for each node {N'|tid(N') = tid(N), N' ∈ ℕ'_A ∪ ℕ'_I}:
5.             make a k-path P = (tid(N_Root), ..., tid(N), ..., tid(N'_Root))
6.             add P to ℙ
7.             if N' ∈ ℕ'_A than
8.                 remove N' from ℕ'_A
9.                 add N' to ℕ'_I
10.            end if
11.        end for
12.    end for
13. end for
14. return ℙ
```

There are several interesting issues in this procedure that we would like to discuss. The first one is that only active and idle nodes takes part in the line 4, where we check if two nodes have the same tuple id. The reason is that when we check only the leaves we automatically prohibit the generation of the k-paths that has been already produced.

Another interesting issue is transferring the node N' from the set of active nodes $\mathbb{N}'_A$ to the set of idle nodes $\mathbb{N}'_I$ (lines 7-9). This is done in order to prevent the unnecessary expanding of two k-trees.

6.3 Make Results

Purpose: to produce new mk-paths.
Input: two sets of k-paths: $\mathbb{P}_{new}$ and $\mathbb{P}$.
Output: prints out the results and outputs a new set $\mathbb{P}$.

```
MakeResults(ℙ_new,ℙ)
1. for each P ∈ ℙ_new:
2.     print all mk-paths {MP|MP = {P, P_1, ..., P_n}, P_i ∈ ℙ}
3. end for
4. add ℙ_new to ℙ
5. return ℙ
```

The purpose of this procedure is to produce new mk-paths by making all possible combinations of each k-path from the newly generated set $\mathbb{P}_{new}$ and all previously made k-paths in a set $\mathbb{P}$. It outputs only these mk-paths that comply with Definition 5.

6.4 Expand

Purpose: to add the children (if possible) for every leaf node in the tree T.
Input: a tree T and a database D.
Output: an expanded tree T.

```
Expand(T,D)
1. for each leave {N|N ∈ ℕ_A, T = (ℕ_A ∪ ℕ_I ∪ ℕ_P)}
2.     R_i = tid(N)
3.     for each relation {R'|∃RL, R',R ∈ RL, R' ∈ D}
4.         for each tuple {R'_i|R'_i is adjacent to R_i}
5.             make a new node N' = (R'_i, N)
6.             if ∀N'' ∈ T  tid(N') ≠ tid(N'') than add N' to ℕ_A
7.         end for
8.     end for
9.     if at least one node was added to ℕ_A then
10.        add N to ℕ_P
11.    remove N from ℕ_A
12. end for
13. return T
```

This procedure takes only active nodes and tries to make children for each of them. We add a new node N' to the k-tree T if there is no other node containing the same tuple identifier. This requirement is defined in Definition 3.

Between lines 9 and 11 there is one interesting issue to discuss. A node N is added to the set of parents $\mathbb{N}_P$ only in case it has got at least one child. In other case it is deleted from a tree.

6.5 Stop Criteria

There is one main condition for stopping the search: if after "Expand" step on all the k-trees no new leaf nodes are added – there is no reason to continue with a search process. This criteria uses several properties. The first one is that if the "Expand" procedure has tried to add new children to all of the k-trees and there are no new children nodes, hence in all of the following iterations it will not produce any new nodes also. From this property follows another one. If at each iteration we have the same number of active and idle nodes we can produce only the same k-paths we produced at last iteration. Without no new k-paths no new results (mk-paths) will be produced. Thus we have to conclude that continuing the search process will not give as any new results and we can stop the search.

7 Application Example

In this section we diagram the work process of SpiderLink algorithm. We use locations of keywords presented in Example 5.2. Each location of keywords has been transfered to a separate k-tree. In Figure 7 we have a picture of k-trees after the "Initialization" step. Dashed lines divides forest to three parts indicating which

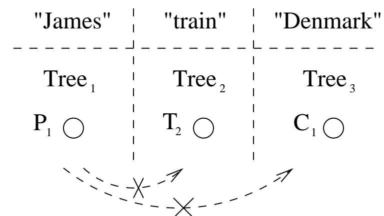


Figure 7: First iteration. Connect step.

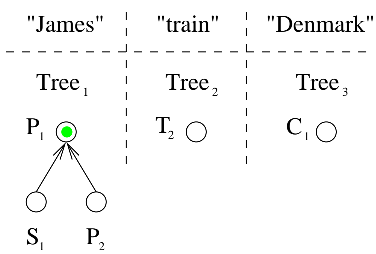


Figure 8: First iteration. Expand step.

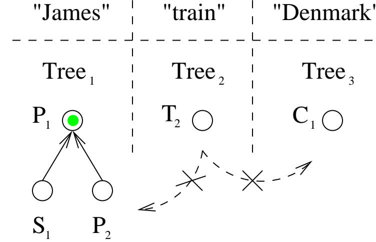
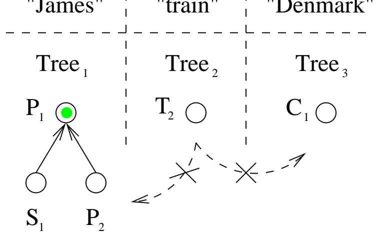



Figure 9: Second iteration. Connect step.

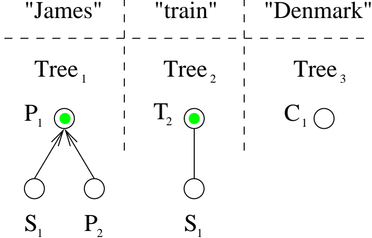


Figure 10: Second iteration. Expand step.

trees belongs to which keywords. We have a forest of three trees, which looks like that:
$forest = \{Tree_1, Tree_2, Tree_3\}$;
$Tree_1 = \{P_1\}_A$; $Tree_2 = \{T_2\}_A$; $Tree_3 = \{C_1\}_A$.
$rootkw(Tree_1)$="James", $rootkw(Tree_2)$="train", $rootkw(Tree_3)$="Denmark".
In all of the following examples we write only a tuple identifier near each node.

Figure 7 shows that "Connect" procedure has no possibilities to build a k-path. There are no leaves having the same tuple identifiers. In Figure 8 there is shown a state of trees after the "Expand" step. Node P_1 in a tree $Tree_1$ became a parent and the are two new active leaves S_1 and P_2.

In the second iteration of algorithm, "Connect" again can not make any k-paths (Figure 9). Figure 10 diagrams the k-trees after the "Expand" step. The same situation is in the third step, where only two new nodes are added to the k-tree $Tree_3$ (Figure 12).

In the fourth step diagramed in Figure 13 "Connect" procedure finds two nodes having same tuple identifiers. It builds a k-path (P_1, S_1, T_2) and a tuple in the k-tree $Tree_2$ becomes idle. This k-path can not be output as a result (as mk-path), because it contains only two keywords. After the "Expand" procedure we have such forest:
$Tree_1 = \{T_2, D_1, D_2, P_3\}_A \cup \{P_1, P_2, S_1\}_P$;
$Tree_2 = \{S_1\}_I \cup \{T_2\}_P$;
$Tree_3 = \{D_1, D_2\}_A \cup \{C_1\}_P$.

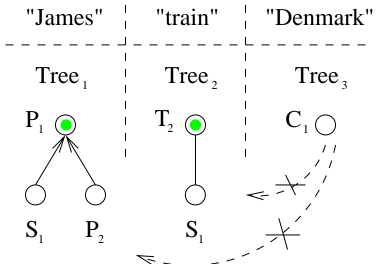


Figure 11: Third iteration. Connect step.

Figure 12: Third iteration. Expand step.

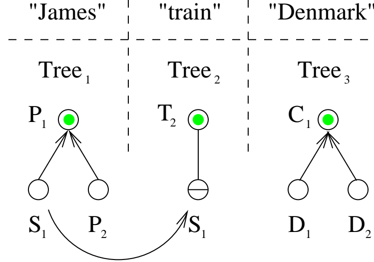


Figure 13: Fourth iteration. Connect step.

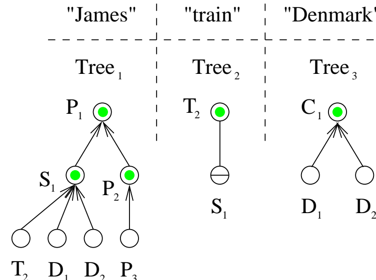


Figure 14: Fourth iteration. Expand step.

Figures 15 and 16 show the fifth iteration of the algorithm. During these two steps k-tree $Tree_2$ is neither connected nor expanded. But at the following iteration two new k-paths are produced (Figure 17): (P_1, S_1, D_1, C_1) and (P_1, S_1, D_2, C_1). These two paths can be combined with the one we have produced previously. Two mk-paths are output to a user: $MK_1 = \{(P_1, S_1, T_2), (P_1, S_1, D_1, C_1)\}$ and $MK_2 = \{(P_1, S_1, T_2), (P_1, S_1, D_2, C_1)\}$. These two answers differ only in one tuple. In one of them there is D_1 in the other D_2. But this difference is important as the results shows "James" connecting to "Denmark" through two different cities: Copenhagen

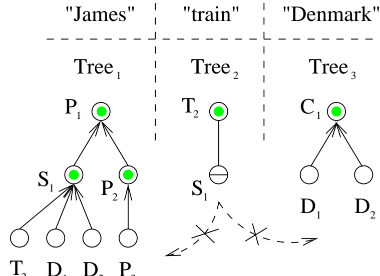


Figure 15: Fifth iteration. Connect step.

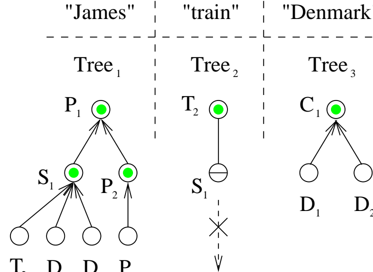


Figure 16: Fifth iteration. Expand step.

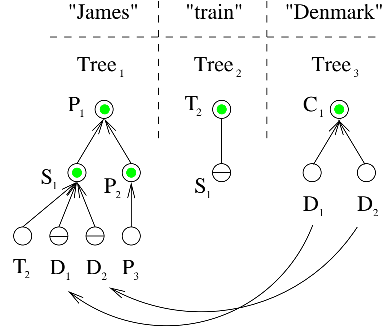


Figure 17: Sixth iteration. Connect step.

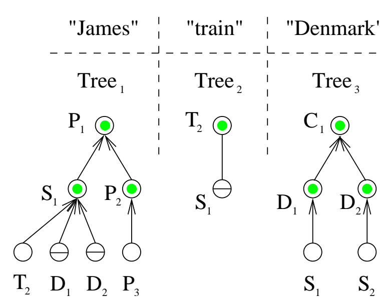
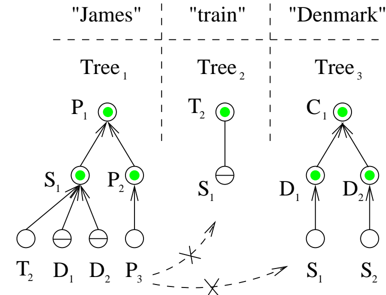


Figure 18: Sixth iteration. Expand step.



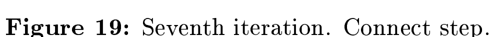


Figure 19: Seventh iteration. Connect step.

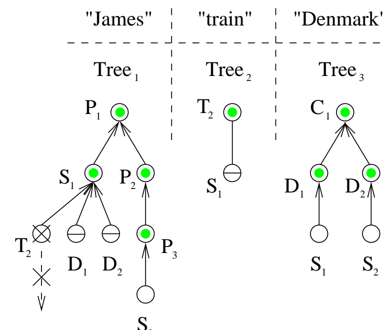


Figure 20: Seventh iteration. Expand step.

and Aalborg. We should notice that two nodes D_1 and D_2 in a k-tree $Tree_1$ became idle.

We have also diagramed the seventh iteration of the SpiderLink algorithm in Figures 19 and 20. In Figure 20 there is shown a new thing. Node T_2 in a k-tree $Tree_1$ can not be expanded, therefore we delete it.

We stop our example here as we have showed all the algorithm processes we intended. The continuing will just repeat "Connect" and "Expand" procedures.

8 Architecture

We have made an implementation of our SpiderLink algorithm. In this section we explain the architecture of this implementation. All implementation was done in JAVA. We used JDBC to connect to the Oracle database.

Figure 21 shows five main components of SpiderLink keyword search engine. In this figure "User" is the starting point of the search process. A keyword query given by user is passed to the tokenizer. The stream is cut to separate tokens and every token becomes a keyword. All the keywords are stored in non capital letters. Tok-

enizer gives a set of keywords to “Keyword location finder”. This module does the work for SpiderLink initialization step. It takes every keyword and scans a database looking for this keyword. “Keyword location finder” issues an SQL query for each keyword for each relation. Lets assume we have keyword KW and we are looking in a relation R. The SQL query looks like that:

SELECT rowid FROM R
WHERE LOWER(A_1) LIKE '%KW%' OR ... OR LOWER(A_n) LIKE '%KW%'.

We use SQL predicate “LIKE” to find exact and not exact matches of keywords. A predicate “LOWER” is used in order to get a keyword match without taking in to account the difference between capital and non capital letters. In the query participate only attributes A_i of type “CHAR” or “VARCHAR”. Searching for keywords only in textual information simplifies the initialization step. The issue of interpreting given keywords as various types of data (like integers, data and so on) is a big and separate problem.

The goal of the SQL queries issued by “Keyword location finder” is to get rowid's of tuples where keyword KW is located in at least one of the values. Using such a query we find exact and partial keyword matches in text values. We use an Oracle database feature of keeping identifier for each tuple. We store tuple id and a relation name in order to make sure that a combination of both elements gives us a unique identifier for a tuple.

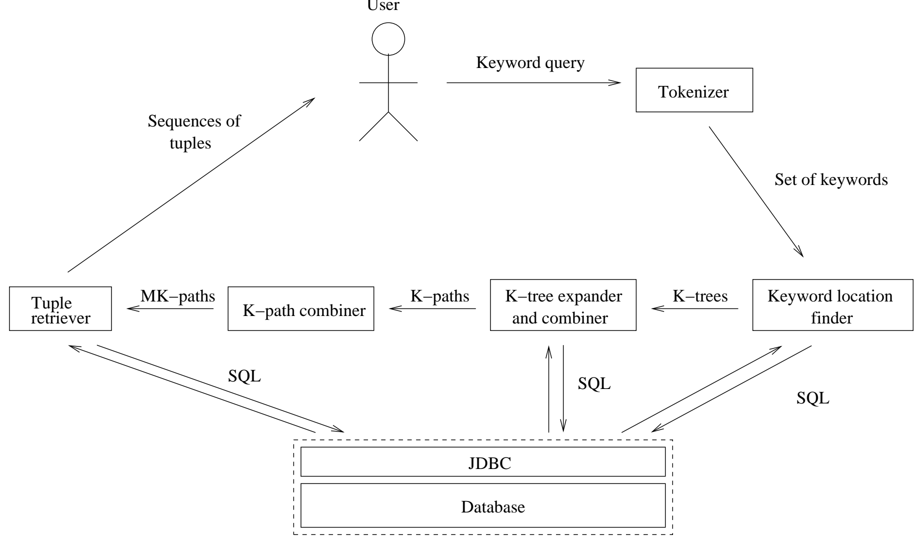


Figure 21: Architecture of SpiderLink implementation

After we find locations for all keywords, we make a k-tree for each location and pass them to “K-tree expander and combiner”. This module is responsible

for generating k-paths. As it is expanding k-trees, it issues an SQL queries to the database. A nice feature of the implementation is that SQL queries are using tuple identifier *rowid*. This is the fastest way to reach the required tuple in a database.

Every time a set of new k-paths is produced, a “K-path combiner” tries to make new mk-paths. If it succeeds, mk-paths are passed to “Tuple retriever”. This last module takes care that a user gets a set of tuples, not a set of tuple identifiers.

An interface we have made is very simple. User launches a compiled JAVA program from a console window. In Figure 22 there is an example of the help information printed in console window. It demonstrates the possibilities of our implementation of algorithm. A user can specify the configuration file and a file with a terms for several sequential queries. User can also decide which type of output it prefers: to get results as a text in console window, or to generate a HTML file with full information from queries. User can also provide the keywords for SpiderLink through the command line. In Figure 23 there is shown an example of simple result output in HTML format.

```
xterm
[atto]> java Main3 -standart --help
dbse [options] [search words]
[options]
  -help, --help      print help
  -cf <file>         use <file> as configuration file
  -sf <file>         use <file> as search parameters file
  -standart          output in stadart mode (defalt output is in HTML mode)

[search words]
  Two or more words should be placed in quotation-marks. E.g. "first second"

More detailed help is available in README file.
```

Figure 22: Example of implemented system input capabilities.

Conclusions

The increasing need for information retrieval search engines for relational databases motivated us to develop the SpiderLink algorithm. In this article we have presented two important issues: the k-tree data structure for keyword searching in relational databases and SpiderLink algorithm which uses k-trees and is a fully developed keyword search engine. In this paper we presented and proved several important properties of k-trees: i) it can be used on databases having single, parallel and hierarchical relationships; ii) it is minimal; iii) it is finite; iv) it can be implemented as a hash-table.

The further development of SpiderLink keyword search system could focus on such issues:

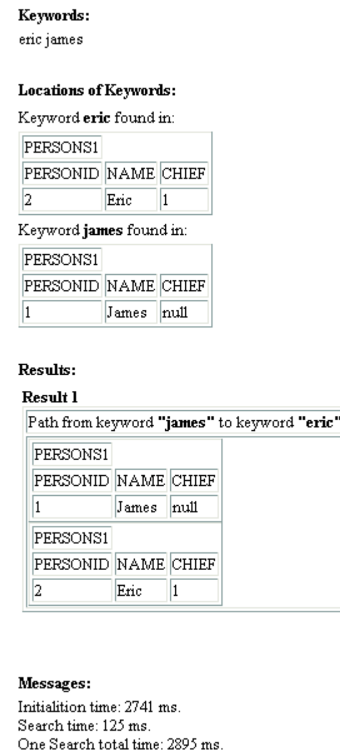


Figure 23: Example of the results in HTML format

- Developing an advanced k-tree data structure that could reuse already gathered information about adjacent tuples when building k-trees.
- Developing a convenient visual representation of keyword search results.
- SpiderLink algorithm can be easily extended to support keyword searches not only on a single database, but also on several databases (treating them as one big database).

References

[1] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A System For Keyword-Based Search Over Relational Databases. In *Proc. ICDE*, 2002.

[2] Valerie S. Allen and Abe Lederman. Searching the deep web - distributed explorit directed query applications. In *SIGIR 2001: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development*

in Information Retrieval, September 9-13, 2001, New Orleans, Louisiana, USA, pages 456–456. ACM, 2001.

[3] Gaurav Bhalotia, Charuta Nakhe, Arvind Hulgeri, Soumen Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in databases using BANKS. In *Proc. ICDE*, 2002.

[4] Michael H. Böhlen, Linas Bukauskas, and Curtis E. Dyreson. The jungle database search engine. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 584–586. ACM Press, 1999.

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[6] Paul Dixon. Basics of Oracle Text Retrieval. *IEEE Data Engineering Bulletin*, 24(4), December 2001.

[7] Roy Goldman, Narayanan Shivakumar, Suresh Venkatasubramanian, and Hector Garcia-Molina. Proximity search in databases. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 26–37, 24–27 1998.

[8] James R. Hamilton and Tapas K. Nayak. Microsoft SQL Server Full-Text Search. *IEEE Data Engineering Bulletin*, 24(4), December 2001.

[9] Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *Proc. 28th Int. Conf. Very Large Data Bases, VLDB*, 2002.

[10] Albert Maier and David E. Simmen. DB2 Optimization in Support of Full Text Search. *IEEE Data Engineering Bulletin*, 24(4), December 2001.