# Query and Answer Models for Keyword Search

**M. Tech. Seminar Report**

Submitted in partial fulfillment of the requirements
for the degree of

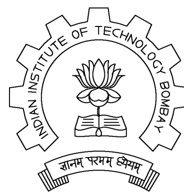
**Master of Technology**

by

**Rose Catherine K.**
**Roll No: 07305010**

under the guidance of

**Prof. S. Sudarshan**




Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

# Acknowledgements

I would like to express my sincere thanks and gratitude to my guide, Prof. S. Sudarshan, for the constant motivation and guidance he gave me through out my seminar preparation, and for having patience to clarify my doubts and for bringing into perspective, the different aspects of the seminar topic. Working with him was a great learning experience.

I would also like to thank Prof. Soumen Chakrabarti, for the clarifications regarding the paper "Optimizing Scoring Functions and Indexes for Proximity Search in Type-annotated Corpora", thereby giving me a good insight into the functioning of the model.

Rose Catherine K.
MTech. 1
CSE, IIT Bombay.

# Abstract

Keyword search is an important paradigm of searching in all domains. It offers the great advantage that the user is not required to know anything regarding how the data is actually organized and stored in the underlying system. When applied to the database setting, where the most common method of querying is by means of query languages, this implies that, the user can simply give a set of terms without specifying the table or column names. But, designing and implementing such a system is not an easy task. It is particularly made difficult by the way databases are designed. The data required to answer a user query, may be split across different tables and columns, due to database normalization. Another major consideration is the form or structure of the answer - it could be in the form of a graph or a tree, or it may be just a tuple or a term. But, it should best convey the information sought by the user. To decide, what can be considered as an answer, in itself, is a challenging task. The semantics of an answer to a multiple keyword query is determined by the system - it could follow OR semantics, AND semantics, or it could be a hybrid of both. Also, in general, there could be many answers to the same query. Then arises the need for ranking them, according to their relevance. The notion of relevance is highly ambiguous, since it is subjective to the user. Many models have been suggested for assigning relevance score to the answers. Some models are in similar lines of the techniques developed in the Information Retrieval area. The success of search engines in the domain of web search has led to the adaptation of such techniques to the database setting, as well. This report is intended to give a description of a few of the keyword search systems, with regard to their query, answer and scoring models.

# Contents

**1 Introduction** **1**
1.1 Representing Data as a Graph . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Keyword Query System Model . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Overview of the report . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 Schema Graph based Search Systems** **3**
2.1 DBXplorer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.1.1 Publish Step . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.1.2 Search Step . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.1.3 Symbol Table design . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.1.4 Answer Model and Ranking . . . . . . . . . . . . . . . . . . . . . . 5
2.2 DISCOVER . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2.1 Data and Answer Model . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2.2 Generation of Answers . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 IR style Keyword search . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Data Graph Based Search Systems** **8**
3.1 BANKS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.1.1 Data Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.1.2 Query and Answer Model . . . . . . . . . . . . . . . . . . . . . . . 8
3.1.3 Ranking answers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.1.4 Bidirectional Search . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.1.5 Activation of Nodes . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.1.6 Response Ranking . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.1.7 Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

**4 Systems for Proximity Search** **12**
4.1 Search incorporating the notion of proximity . . . . . . . . . . . . . . . . . . 12
4.1.1 Data Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
4.1.2 Query Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
4.1.3 Proximity and Scoring Models . . . . . . . . . . . . . . . . . . . . . 13
4.2 Object Rank System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.2.1 Data Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.2.2 Random Surfer model for Ranking . . . . . . . . . . . . . . . . . . . 15
4.2.3 ObjectRank Index creation . . . . . . . . . . . . . . . . . . . . . . . 16
4.3 Proximity Search in Type-Annotated Corpora . . . . . . . . . . . . . . . . . . 17
4.3.1 Data Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.3.2 Query and Answer Model . . . . . . . . . . . . . . . . . . . . . . . 18
4.3.3 Scoring Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.3.4 Learning the Scoring Function . . . . . . . . . . . . . . . . . . . . . 18

**5 Searching using Graph Patterns** **19**
5.1 FleXPath . . . 19
5.1.1 Tree Pattern Query . . . 19
5.1.2 Answer to a Tree Pattern . . . 20
5.1.3 Ranking Scheme and Predicate Penalty . . . 20
5.2 The NAGA system . . . 21
5.2.1 Data Model . . . 21
5.2.2 Query and Language Model . . . 22
5.2.3 Answer Model . . . 23

**6 Conclusion and Future Work** **24**

# 1 Introduction

The amount of data that is available today is incomprehensible, and is increasing at an alarming rate. It is stored in a variety of forms like web pages, XML data, relational data, etc. Searching for the right piece of data has become a very important task of everybody's life. Web search has already become a multi-billion dollar industry. The next important search domain is that of Relational databases. Relational databases are used world over, to store huge volumes of data, like enterprise data, company archives, etc. To retrieve this data, at present, structured query languages are used. But to ask suitable queries, the user needs to know the schema that has been used to store the data. To avoid this, and to enable searching the database through the internet, the most commonly used technique is to provide customized forms. However, using forms has its own limitations - different forms will have to be provided for each of the tables, views etc. Also, these forms cannot be completely independent of the underlying schema.

The above facts and constraints served as a motivation to find another paradigm to search and retrieve data stored in databases. This quest lead to "Keyword Searching", which is an unstructured method of querying and which has already been popularized by the web search engines. The greatest advantage of this is that the user requires no knowledge of the underlying database schema.

It is a challenging task to enable keyword search in databases. The techniques used by web search engines cannot be directly applied to the database setting. Here, due to database normalization, the information required to answer a keyword query may be split across tables. Hence, several table joins may have to be done, on the fly. In addition, databases has some unique characteristics, like different types of edges, attributes of nodes, semantics associated with tables etc., which can be used to improve the quality of the keyword search. Also, the physical database design, like the availability of indexes on certain columns, have to be taken into account, for efficient performance of the keyword search.

Another important aspect of answering keyword queries is that, the answers must be generated incrementally, in the order of their relevance. Usually, the users are interested only in the first few answers. Hence, it will be a waste of resources and time, to generate all the answers and then, sort them for presenting them finally to the user. For this reason, most of the keyword search systems also incorporate, what is called as the 'Top-k query processing' techniques, to improve their efficiency. But, in this report, this aspect is not discussed in detail.

## 1.1 Representing Data as a Graph

To enable keyword searching on the data (both structured and semi-structured), it has to be represented in a suitable form. The most popular representation adopted is the graph model. There are two kinds of graphs used by the keyword search systems - the Schema Graph and the Data Graph.

(i) A Schema Graph describes the schema of the data. It can be considered as a meta-level representation of the data, since it gives details about the data (but not the data itself). It also, constraints the edges that are permissible in the data graph. i.e., every edge in a data graph has an equivalent edge in its schema graph. For a relational database, schema

graph is constructed as follows: the tables in the database form the nodes. Edges between two nodes, capture some relationship or constraint, between their corresponding relations.

(ii) A Data Graph can be considered as an instantiation of its schema graph, such that, it conforms to the latter with respect to the types of nodes and edges, and the constraints specified in terms of edges that are allowed. The data graph contains actual data which is split across different nodes and edges. For a relational database, the commonly used construction is as follows: the tuples of the database form the nodes and the cross references between them, like foreign key references, inclusion dependencies, etc., form the edges of the graph. The nodes may also be a cell in the table, according as the granularity required.

There is also a concept of node weight and edge weight. The exact meaning of these depend on the semantics of the system. In general, the node weight could correspond to its importance or popularity in the entire collection. The edge weights could be used to represent the similarity or the distance between nodes that they connect, or the fraction of the node weight that can be transferred from a node to another.

Representing data as a graph has several advantages. It reveals the high level structure and the inter-relationships among various components. It also allows the use of graph based algorithms for traversing the data and producing answers according to the semantics followed. Answers could be generated in the form of graphs or trees, or they could be just tuples or terms. Answers will contain nodes that have the keywords. But, it may also contain nodes that don't have any keywords, according to whether, that node is required to make the answer graph connected or not. But, all of this is specific to the system.

## 1.2 Keyword Query System Model

This section describes, in general terms, the framework of a keyword query system. A keyword query system is a complex system which is capable of taking as input, a set of words, called keywords and give an appropriate answer. Here, the structure and semantics of the answer is specific to the query answer system. A system for keyword query consists of the following:

(i) Data Model: It describes the high-level representation of the data in the system, such that it reflects the constraints, associations, and organization of the data. The actual implementation of the representation is not of concern, here. The most popular form of representation is the graph model, which was described in the Section 1.1.

(ii) Query Model: It specifies the structure of the input that can be given to the system. For keyword queries, the most common form of input is a set of words or terms. This simplifies the task of querying, since, the user is required to know neither any query language nor the schema of the database. A more powerful form of querying is by using graph or tree patterns. This form of representation enables the user to specify the constraints which the answer must satisfy. Specific examples of such systems are discussed in Section 5.

(iii) Answer Model: It specifies the structure of an answer to a query and the requirements that it must satisfy according to the semantics of the system. The answers are usually represented as a graph or tree, or it may be just a tuple or a term.

(iv) Scoring Model: In general, there will be many answers to the same keyword query and hence most of the systems employ a scoring model, which assigns a score to each of the answers, based on their relevance. The notion of relevance is very ambigous, since it depends on the user. Also, since keyword querying is not a powerful method in terms of expressiveness, the users are unable to articulate their requirements exactly. Hence, instead of a single answer, the system must return top few documents with the highest scores. The score is dependent on the semantics of the system. A simple method used, is to give higher score to an answer with smaller number of joins. But, most systems use complex rules to assign scores, to improve the quality of the top ranked answers.

### 1.3 Overview of the report

This report is intended to give an overview of different keyword search systems, focusing on systems that support keyword search on databases. The report has been organized as follows: the first section gives an introduction to the paradigm of keyword search, bringing to notice, its relevance at present. It also gives a general description of the high-level structure of keyword search systems. In the following sections, different search systems are discussed, concentrating more on their query, answer and scoring models. The search systems have been grouped roughly into four categories - Schema Graph and Data Graph based search systems, Systems for Proximity search and Graph-pattern based search. But this categorization is quite vague, since, all systems have many features in common, but at the same time, are quite different from each other. Many of the systems can even belong to more than two sections; but for the purpose of the report, they have been placed in a category, that they seem to fit into, the best. Of these search systems, BANKS and ObjectRank have been discussed in detail, with respect to their data, query, answer and scoring models. A brief description of the algorithms they use, is also given. Finally, the report concludes with a summary of the systems discussed and a glimpse of the road ahead.

## 2 Schema Graph based Search Systems

Schema graph, as mentioned before, is one of the preferred forms of representation of data. An important purpose of it is to guide the direction of the search and thus, optimize the search. In this section, a few systems that employ schema graph in their search, are discussed.

### 2.1 DBXplorer

Given a set of query keywords, DBXplorer [ACD02] returns all rows (either from single tables, or by joining tables connected by foreign-key joins) such that the each row contains all keywords. Enabling such keyword search requires (a) a preprocessing step called Publish that

enables databases for keyword search by building the symbol table and associated structures, and (b) a Search step that gets matching rows from the published databases.

### 2.1.1 Publish Step

A database is enabled for keyword search through the following steps:

(i) The set of tables and columns of the database to be published, are identified.

(ii) Auxiliary tables for supporting keyword search are created. The most important structure is a symbol table S which stores the locations, in terms of the tables, columns or rows of keywords in the database.

### 2.1.2 Search Step

Input to this step is a query consisting of a set of keywords. It is answered as follows:

(i) The symbol table is looked up to identify the location of query keywords in the database.

(ii) All potential subsets of tables in the database that, if joined, might contain rows having all keywords, are identified and enumerated. A subset of tables can be joined only if there is a sub-tree called a join tree in the schema graph that contains these tables as nodes.

(iii) For each enumerated join tree, a SQL statement is constructed that joins the tables in the tree and those rows that contain all keywords are selected.

(iv) The final rows are ranked and presented to the user.

Steps (ii) - (iv) are described in detail in Section 2.1.4

### 2.1.3 Symbol Table design

An important consideration in the symbol table design is deciding the granularity of the location of the keyword in the database. Commonly employed granularity levels are:

(i) Column level granularity (Pub-Col): For every keyword, the symbol table maintains the list of all database columns that contain it. Hence, the entries are lists of table.column values.

(ii) Cell level granularity (Pub-Cell): Here, for every keyword, the symbol table maintains the list of database cells that contain it. Therefore, the lists contain table.column.rowid values

The Pub-Col symbol table design is usually preferred to the Pub-Cell table design, since the former outperforms the latter in terms of (a) space and time requirements for building the symbol table, (b) effect on keyword search performance and (c) ease of symbol table maintenance, which are the three most important considerations with respect to performance. But, a Pub-Cell design will have to be adopted if certain columns do not have indexes. In such cases, a hybrid symbol table design is employed where the column contents of those columns for which an index is available is published with Pub-Col granularity and others are published with Pub-Cell granularity.

### 2.1.4 Answer Model and Ranking

The answers for a query $K_1, K_2, ..., K_k$, where $K_i$'s are the keywords, are rows (which could be from a single table or obtained after join of multiple tables) that contain all the keywords.

**Enumerating Join Trees :** The objective of this procedure is to find out that tables of the database, which on joining will yield tuples containing all the keywords of the query.

Let $MatchedTables$ be the set of tables that contain at least one of the query keywords. Then, this procedure enumerates join trees which are sub-trees of the schema graph $G$, such that: (a) the leaves belong to $MatchedTables$ and (b) together, the leaves contain all keywords of the query.

The algorithm for enumerating join trees assumes that $G$ is a tree. Nodes representing the $MatchedTables$ set are coloured black, while the rest are white. First $G$ is pruned by repeatedly removing white leaves, until all leaves are black. The resulting tree, $G'$, is guaranteed to contain all potential matching join trees. To enumerate all qualifying sub-trees of $G'$, pick the keyword that occurs in the fewest black nodes of $G'$ and do breadth-first enumeration of all sub-trees of $G'$ starting from it. But, if $G$ cannot be assumed to be a tree, then, the join tree enumeration involves bi-connected component decomposition of $G$, followed by the enumeration of join trees on a possibly cyclic schema graph.

**Searching for Rows:** Given a set of join trees, this procedure maps each join tree to a single SQL statement that joins the tables as specified in the tree, and selects those rows that contain all keywords. The generated SQL statement will have selection conditions on columns for a Pub-Col symbol table and rowids for a Pub-Cell symbol table. This is the only stage of the search where the database tables are accessed.

**Ranking:** The approach followed is to rank the rows by the number of joins involved. This is based on the intuition that, larger is the number of joins involving many tables, harder are the results to comprehend and hence, of lesser relevance.

## 2.2 DISCOVER

For a keyword query, DISCOVER [HP02] returns qualified joining networks of tuples, which are sets of tuples that are associated because they join on their primary and foreign keys and collectively contain all the keywords of the query. There are two major steps involved in answering. First, all join expressions that can give potential answer graphs, are generated. Then, plans for the efficient evaluation of this set of candidate graphs are generated. This exploits the presence of common subexpressions in the candidates, for efficiently generating the qualified joining networks of tuples.

### 2.2.1 Data and Answer Model

DISCOVER uses a schema graph as well as a data graph for representing data, the structure of both of which were discussed in Section 1.1. The answer to a keyword query is also a graph. A subgraph of the database graph is considered as an answer, if it satisfies the following requirements:

(i) Total: Discover follows the AND semantics for the keywords and hence, any answer graph should contain ALL the words in the query.

(ii) Minimal: If we remove any node from the answer graph, then either, it becomes disconnected or it is no longer total. This notion is described formally below. Minimality requirement ensures that the answer graphs generated are not redundant.

**Joining Network of Tuples** $j$ is a tree of tuples where for each pair of adjacent tuples $t_i, t_j \in j$, where $t_i \in R_i, t_j \in R_j$ , there is an edge $(R_i, R_j)$ in the schema graph and $(t_i \bowtie t_j) \in (R_i \bowtie R_j)$. Here, $R_i$ is a relation in the database. When each internal node of the joining network has exactly two neighbours, then it is called a *joining sequence of tuples*. This situation occurs when the query contains only two keywords. The answer generated by the system is a *Joining Network of Tuples* that satisfy the above mentioned requirements of Totality and Minimality, and are called Minimal Total Joining Networks of Tuples (MTJNT).

The system also uses a *Master Index* which when queried with a set of keywords $k_1, ..., k_m$, outputs a set of tuple sets $R_i^{k_j}$ for each of the relations $R_i$ in the database and for $j = 1, ..., m$. $R_i^{k_j}$ consists of all tuples of relation $R_i$ that contain the keyword $k_j$ and is called a basic tuple set.

### 2.2.2 Generation of Answers

This section gives a high level description of how DISCOVER generates MTJNTs for a given keyword query $k_1, ..., k_m$. The main steps are as given below:

(i) The Master Index is used to get the basic tuple sets corresponding to the terms in the query.

(ii) The basic tuple sets are passed to a Tuple Set Post-Processor, which produces tuple sets $R_i^K$ for all subsets $K$ of $\{k_1, ..., k_m\}$. The tuple set $R_i^K$ contains the tuples of $R_i$ that contain all keywords of $K$ and no other keywords.

(iii) These tuple sets along with the schema graph of the database are passed to the *Candidate Network Generator*. Now, define a *Joining Network of Tuple Sets J* as a tree of tuple sets where for each pair of adjacent tuple sets $R_i^K$ , $R_j^M$ in J, there is an an edge $(R_i, R_j)$ in the schema graph. The *Candidate Network Generator* generates only those *Joining Network of Tuple Sets* which can give an instance of the database, that is a MTJNT. The output of the generator is called *Candidate Network*

(iv) The set of candidate networks is then, passed to the *Plan Generator*, which outputs an execution plan, as a list of joins to be performed on the database tables. The generator, also creates intermediate tables which can be reused later, in computing joins and thus optimizing the evaluation of the network.

(v) The last step is to pass the plan to the *Plan Execution Module* which generates SQL statements and returns the result of their execution on the database.

DISCOVER gives higher score to smaller MTJNTs and hence, the results returned by the *Plan Execution Module* is in the order of the number of joins involved in their computation.

## 2.3 IR style Keyword search

Hristidis et. al. suggested a database search system in [HGP03] which incorporates IR techniques of proximity, in answering keyword queries on a database. Contemporary RDBMS possess efficient querying capabilities for text attributes, but they require the user to specify the column for search. The main idea proposed in the paper is to use these features of the underlying RDBMS, to efficiently process a keyword query. The data and query model followed is same as that given in [HP02], which has been discussed in Section 2.2. The answer to a keyword query is a *Joining Tree of Tuples,* $T$ which is defined as a tree of tuples where each edge $(t_i, t_j)$ in $T$ , where $t_i \in R_i$ and $t_j \in R_j$ is such that $R_i$ and $R_j$ are elements of the schema graph $G$, and $t_i \bowtie t_j \in R_i \bowtie R_j$. Also, the answer must satisfy the totality and minimality conditions specified in Section 2.2.1.

**Scoring Model:** Let $T$ be the joining tree of tuples for which a score needs to be assigned. The first step is to assign a relevance score to each of the textual attribute $a_i \in T$. This single-attribute score with respect to the query $Q$ is determined by the IR engine employed in the underlying Relational Database. Once this is done, the single-attribute scores are combined using a function *Combine*, to give the final score of $T$.

An example of IR definition for a single-attribute scoring function is :

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

where, $w$ a term that is present in both $Q$ and $a_i$. $tf$ gives the term frequency of $w$ in $a_i$, $df$ gives the document frequency, which is the number of tuples in the relation to which $a_i$ belongs, that has $w$ in this attribute. $dl$ gives the size of $a_i$ in characters, $avdl$ is the average of $dl$ values across $a_i$'s, $N$ represents the total number of tuples in the relation, and $s$ is a constant.

The final score of the tree $T$ is obtained by combining its single-attribute scores. Let $A = < a_1, ..., a_n >$ be a vector with all textual attribute values for T . Then the score of $T$ for $Q$ is defined as:

$$Score(T, Q) = Combine(Score(A, Q), size(T)), where$$

$$Score(A, Q) = < Score(a_1, Q), ..., Score(a_n, Q) > and$$

$$Combine(Score(A, Q), size(T)) = \frac{\sum_{a_i \in A} Score(a_i, Q)}{size(T)}$$

**AND, OR semantics:** The system incorporates AND and OR semantics in the following way:

(i) AND semantics: Any tuple tree that does not contain all the query keywords is assigned a score of 0. But, if all the keywords are present, then the score is given by *Combine* function.

(ii) OR semantics: For all tuple trees, the score is given by the *Combine* function.

The *Combine* function described above can be chosen appropriately to calculate the relevance according to the semantics of the system. This together with the AND or OR semantics specified by the queries enables the system to identify tuple trees with highest relevance.

# 3 Data Graph Based Search Systems

Data Graph representation, as mentioned before, is one of the methods for representing data. In almost all the applications, the data graph is so huge that it no longer fits in the main memory. A major concern in this scenario is the efficiency of search and extraction of best answers. Hence, the algorithms employed, must be carefully designed to avoid long delays. In the sections below, the BANKS system, which is one of the highly regarded search systems in the Database Keyword Query community, is discussed in detail.

## 3.1 BANKS

This section is aimed at giving a detailed description of BANKS. It stands for Browsing ANd Keyword Searching. In addition to keyword searching, it also allows users to browse the schema of the database and decide interactively, the view of the database that they want. In the sections below, the system has been explained with regard to its data, query, answer and ranking models. An improved version of BANKS has been proposed in [KPC+05] and is discussed in Section 3.1.4.

### 3.1.1 Data Model

BANKS system [BHN+02] models the relational database as a graph. The tuples of the database constitute the nodes of the graph and the foreign key - primary key relationship between the tuples form the edges, which are directed. Each node has a node-weight associated with it, which is a measure of its prestige. This weight is a function of the node's indegree - higher the node-weight, higher is its prestige. Edges too have weight, which is a measure of the proximity of the nodes that it connects. Hence, based on the semantics, different types of edges can be given different weights. Larger weight corresponds to lower proximity. For example, an edge between a paper and its author could have a smaller weight than an edge between a paper and another paper that it cites.

Describing edge weights formally, let $u$ and $v$ be nodes in the database graph and $R(u)$ and $R(v)$ be the relations they belong to. Let $s(R(u), R(v))$ denote the similarity between the two relations. Also, $IN_v(u)$ is the in-degree of $u$ contributed by tuples belonging to relation $R(v)$. Then, the weight of the directed edge $(u, v)$ depends on whether there exists links between $u$ and $v$ in the graph, and is defined as follows: if the graph contains only $(u, v)$, the weight of $(u, v)$ is set to $s(R(u), R(v))$. Else if only $(v, u)$ exists, then, weight of $(u, v)$ is set to $IN_v(u) \times s(R(v), R(u))$. If both the links exists, then, the weight is set to the minimum of the two.

### 3.1.2 Query and Answer Model

A query consists of terms $t_1, t_2, ..., t_n$, where $n \geq 1$. The answer to the query is a directed rooted tree containing all the keywords, and is called a connection tree. The root is called the information node and is a common vertex from where there exists path to all the keyword nodes, which are the leaves of the tree.

To find an answer, first job is to locate nodes of the database graph that are relevant to the query. BANKS considers a node to be relevant if it contains any one of the search terms

as a part of an attribute value or meta data (like column names). Now, define $S_i$ to be the set of nodes that are relevant to the term $t_i$. Hence, the answer tree must contain atleast one node from each of $S_i$. Since, there will be many trees that are potential answers, they must be presented to the user in the order of their relevance. Assigning relevance score to the answer trees is discussed in Section 3.1.3.

**Searching for the Best Answers:** Answer trees, as mentioned before, consists of nodes that contain the keywords of the query. But, they may also contain nodes that don't have any of the search terms, according as whether they are required to make the tree connected or not, and hence they are Steiner trees. But, since computation of minimum Steiner trees is NP complete, BANKS uses the Backward Expanding Search Algorithm, which has two main advantages:

(i) It enables generation of not just the most relevant tree, but also, other trees with high scores, since they too could be part of what the user is searching for.

(ii) It offers heuristic solution for incrementally computing query results.

As defined before, $S_i$ denotes the set containing all the nodes that are relevant to the search term $t_i$. Also, define $S = \bigcup S_i$. Then, the backward expanding search algorithm concurrently runs $|S|$ copies of Dijkstra's single source shortest path algorithm, where, each copy corresponds to a node in $S$, and starts from that node as the source. The copies are run concurrently by creating an iterator interface to the shortest path algorithm.

Each copy traverses the graph in reverse direction. The basic idea is to locate a node from which there is a forward path to atleast one node in each of $S_i$. In each iteration, the algorithm picks that iterator, whose next vertex to be output is at the least distance from the source node.

To find information nodes and the corresponding connection trees incrementally, the following procedure is adopted:

(i) Within each vertex $v$, a nodelist $v.L_i$ is maintained for each search term $t_i$.

(ii) Consider an iterator whose source node is $u \in S_i$ visiting $v$. To generate the new connection trees containing node $u$, take the cross product of node $u$ with the rest of the nodelists $\left\{ u \times \prod_{i \neq j} v.L_j \right\}$ and each cross product tuple corresponds to a connection tree rooted at node $v$.

(iii) After generating all connection trees, $u$ is inserted in to $v.L_i$.

A less pronounced deficiency of the algorithm is that the trees may not be generated in the exact order of decreasing relevance. This is because, it doesn't consider the node weights while generating the connection trees, whereas, the relevance of a tree depends on both the node and the edge weights. But generating all connection trees and then sorting them to get the best answers would increase computation costs. To avoid these overheads, BANKS apply the following heuristic: Maintain a small fixed-size heap of generated connection trees, ordered on the relevance of the trees. Add trees to the heap as they are generated. When a new tree needs to be added and the heap is full, output the tree of highest relevance and replace it with the new one, in the heap.

### 3.1.3 Ranking answers

The approach followed by BANKS, to rank the answer trees, is analogous to the techniques developed in web search. There is a notion of proximity between nodes which is measured by the weight of the edges connecting them. Also, each node has a prestige value associated with it, which is a measure of its popularity and is independent of the query. Prestige is defined as a function of the indegree of the node - if a larger number of nodes refer to a particular node, then the latter can be assumed to be important, in general.

The relevance score of an answer tree is formed from two quantities - its overall node score, $Nscore$, and its overall edge score, $Escore$. They are defined as given below:

- $Nscore$ of a tree is defined as the average of the node scores, $Nscore(v)$ of the leaf nodes (containing the keywords) and the root node. Here, the individual node score $N(v)$ of a node $v$, is normalized to obtain $Nscore(v)$ using $N(v)/N_{max}$ or $log(1+N(v)/N_{max})$, where $N_{max}$ is the maximum node weight in the graph. The latter formula may be preferred to the former, since it depresses the score more and hence reduces the effect of any outlier. A node containing more than one search terms is counted as many times as the number of search terms it contains. This helps to avoid giving extra weight to trees with separate nodes for each keyword.
- $Escore$ of a tree is defined as $Escore = 1/(1+\sum_{e} Escore(e))$. Here, $Escore(e)$ is obtained by normalizing the scores $w(e)$ of individual edges, $e$, using either of $w(e)/w_{min}$ or $log(1+w(e)/w_{min})$, where $w_{min}$ is the minimum edge weight in the graph. It can be seen that, the formula for $Escore$ of a tree, gives lower relevance to larger trees.

The overall relevance score can be an additive or multiplicative combination of the two scores. The additive combination uses the formula $(1-\lambda)Escore + \lambda Nscore$ , and the multiplicative combination uses the formula $Escore \times Nscore^{\lambda}$, where, $\lambda$ is a factor controlling their relative weightage.

### 3.1.4 Bidirectional Search

The Backward Search algorithm discussed in the previous section, has many drawbacks in terms of efficiency and space requirements. Its performance will suffer if some keyword matches a large number of nodes or, if some node that it expands, has a large degree. Also, it will generate a large number of iterators, if a keyword matches a large number of nodes. A better algorithm for searching in the graph representation of data, called Bidirectional Search, has been proposed in [KPC+05]. The general idea is that, instead of searching backwards from all the keyword nodes, search backwards only from a select number of nodes. This gives potential root nodes, from which, search can proceed in the forward direction, for the rest of the keyword nodes.

Major differences between this approach and the former model of BANKS, has been outlined below:

(i) Prestige of each node is computed based on a biased random walk, similar to the computation of global ObjectRank (described in Section 4.2.2 ), except that, here, the probability

of moving along a particular edge is inversely proportional to its edge weight, taken from the data graph.

(ii) Nodes have activation with respect to the keywords. This is used to prioritize the frontier for searching. Node activation is discussed in Section 3.1.5.

(iii) Here, there are only two iterators: the *incoming iterator* and the *outgoing iterator*. The *incoming iterator* replaces all the shortest path iterators of the Backward Search algorithm. It chooses the node with the highest activation, to be explored next. The *outgoing iterator* starts from the nodes explored by the *incoming iterator* and moves along forward edges.

### 3.1.5 Activation of Nodes

To improve the efficiency of the search, the notion of activation of nodes is introduced. The idea is that, nodes containing keywords are considered to be active and they spread this activation through the edges, to their neighbouring nodes to make them too, active. The activation undergoes damping with distance or weight of the edge and hence, the neighbours receive only a fraction of the activation. The *incoming iterator* chooses that node with the current highest activation, for expanding next. This leads to prioritization of the frontier, thus preferring paths with less branching and of lesser weight, giving better answers.

***Activation Initialization:*** Let $t_i$ be a keyword in the query and let $S_i$ be the set of nodes in the datagraph, which contain $t_i$. Then, all nodes $u \in S_i$ are added to the incoming iterator, with the initial activation given by

$$a_{u,i} = \frac{nodePrestige(u)}{|S_i|}$$

Thus, keyword nodes with larger prestige have higher activation. At the same time, if the node corresponds to a keyword that matches a large number of nodes, then the activation of the individual nodes are reduced.

***Activation Spreading:*** Each node transfers only a fraction $\mu$ of its activation and retains the remaining $1 - \mu$ fraction. As mentioned before, the activation transferred from a node to its neighbours, suffers damping, which increases with the weight of the edge connecting them. The fraction of activation transferred, is alloted to the neighbours as follows: for the *incoming iterator*, it is distributed amongst nodes to which there is an edge from this node; for the *outgoing iterator* it is distributed amongst nodes from which, there is an edge to this node. Both the allocations are done in inverse proportion of the edge weight.

A node may receive activations for the same keyword, from more than one neighbours. In such cases, the activation of that node, with respect to that keyword is defined as the maximum of the amount received. Now, the overall activation of the node is defined as the sum of its activations for the individual keywords.

### 3.1.6 Response Ranking

The ranking of answers is quite similar to the model discussed in Section 3.1.3. The important points with respect to the scoring model are as below:

(i) For a forward edge $(u, v)$ with weight $w_{uv}$, the corresponding backward edge $(v, u)$ has weight $w_{vu} = w_{uv} log_2(1 + indegree(v))$.

(ii) The score for an answer tree $T$ with respect to keyword $t_i$, $s(T, t_i)$ is defined as the sum of the edge weights on the path from the root of $T$ to the leaf containing that keyword.

(iii) Now, the aggregate edge-score $E$ of $T$ is defined as $\sum_i s(T, t_i)$.

(iv) The tree node prestige $N$ is defined as the sum of the node prestiges of the leaf nodes and the answer root.

(v) The overall tree score is defined as $EN^\lambda$ where $\lambda$ controlls their relative weightage.

### 3.1.7 Comparison

The authors have compared the performance of the new version of BANKS that employs the Bi-directional Expansion search, with that of the earlier version. An overview of some of the important findings, is given below:

(i) An important metric is the number of nodes explored and the number of nodes touched, to retrieve all the relevant results. This is indicative of how effective the system was, in discriminating the paths to be explored. The node-counts of Bi-directional search system was around two orders of magnitude lesser than that of the Backward search system .

(ii) Another major issue is the time required to output the relevant results. In this too, the Bi-directional search system beat the Backward search system, by atleast an order of magnitude. This result and the previous were for a modified version of the backward algorithm, were only one backward iterator was used. This was to seperate out the effect of using only a single iterator for the Bi-directional search.

# 4 Systems for Proximity Search

Proximity search is much beyond simple keyword match. It also takes into account, the distance between the keywords, in the match. Keyword matches beyond a particular distance can be ignored by assuming that, it doesn't reveal any relationship between them. In some text search systems, it also considers the order in which the keywords appear. It is very useful when the answer sought is an object which is close to a set of specified keywords. In this section, few such search systems are discussed.

## 4.1 Search incorporating the notion of proximity

Goldman et. al. suggests a proximity based model in [GSVGM98], which is one of the earliest works in the field of keyword search in databases. This model is motivated by the ranked retrieval done by Information Retrieval engines that rank documents by taking in to consideration, amongst other criteria, the textual proximity of keywords within the documents. When applied to the database graph, proximity is measured as the shortest distance between the nodes.

#### 4.1.1 Data Model

The data is modeled as a data graph. Edges are given different weights to reflect the semantic closeness of the nodes that it connects. Smaller weight indicates more closeness. Once the database has been modeled as a graph, proximity of one node to another can be found out as the length of the shortest path connecting them.

For a disk-based graph, if Dijkstra's Single Source Shortest Path Algorithm is applied, finding object distances will need to access the data from the database frequently. This hurts the efficiency of the search system. To diminish this effect, it would be better to compute beforehand all the shortest pair distances. But, obviously, this does not scale with the database size. The paper proposes a better approach of computing object distances using a method of indexing, called Hub Indexing Scheme.

***Hub Indexing Scheme*** consists of the following:

(i) Hub Set $H$: It is a set of vertices, which when removed, will decompose the graph into two disjoint sub-graphs. The shortest distance between each pair of hubs is also known.

(ii) A table of distances: It stores the shortest distance between pairs of nodes whose shortest path does not cross through the elements of $H$. This also includes the shortest distance between the nodes and those in the hub set.

Suppose if removing the hubs disconnects the graph into sub-graphs $A$ and $B$, then, using hub index requires to store only $|A| + |B|$ distances, instead of $|A| \times |B|$. The basic idea behind this scheme is as follows: Suppose $a$, $b$ are the two objects whose shortest inter-object distance we wish to compute. If both of them belong to the Hub Set or the same sub-graph, i.e., $A$ or $B$, then the distance is already available in the table. In the other case, suppose they belong to different subgraphs. Without loss of generality, let $a \in A$ and $b \in B$. Then the shortest path between them passes throught the hub set. Then, choose two hub vetices, $p$ and $q$, such that $d(a,p) + d(p,q) + d(q,b)$ is the smallest. Here, $p$ and $q$ can be the same vertex also. Now, this quantity can be computed easily, since all the distances are readily available in the table.

#### 4.1.2 Query Model

Proximity searches are specified by a pair of queries: (a) Find Query: This specifies the type of the answer to be retrieved. For e.g., the user can specify that the query is for objects of type "movie". Hence, it defines a set $FindSet$ of objects that can potentially be the answer. (b) Near Query: This specifies the keywords that define a $NearSet$.

The basic idea is to rank the objects in the $FindSet$ based on their proximity to the objects in the $NearSet$.

#### 4.1.3 Proximity and Scoring Models

Let $F$ be the $FindSet$ and $N$, the $NearSet$. Also, let $r_F(f)$ denote the ranking of $f$ in $F$ and $r_N(n)$, that of $n$ in $N$. The distance between any two objects $f \in F$ and $n \in N$ is taken as the weight of the shortest path $d(f,n)$ between them in the graph. Now, the *bond* between $f$ and $n$, for $f \neq n$ is defined as:

$$b(f,n) = \frac{r_F(f) r_N(n)}{d(f,n)^t}$$

Here, $t$ is the tuning component that decides the impact of distance on the bond.

Following scoring functions have been suggested to rank each of the $FindSet$ objects based on their bonds with all of the $NearSet$ objects.

(i) Additive : $score(f) = \sum_{n \in N} b(f,n)$

(ii) Maximum : $score(f) = max_{n \in N} b(f,n)$

(iii) Beliefs : $score(f) = 1 - \prod_{n \in N} (1 - b(f,n))$

Based on the function used, the score of each of the $FindSet$ objects are found out. They are then ranked in the order of decreasing scores and presented to the user.

## 4.2 Object Rank System

This section is intended to give a detailed description of the Object Rank system [BHP04], which is yet another important keyword search system for databases. The Object Rank system adapts the notion of PageRank [BP98] to suit the database setting, by introducing the concept of authority. Nodes having the terms of the query are given some authority and this authority is trasferred by them to their neighbours in a fixed manner. Each node accumulates the authority transferred to it by its neighbours, to give their final ranking with respect to the query. Details of the approach is discussed in Section 4.2.2.

### 4.2.1 Data Model

ObjectRank models database as graphs. There are four graphs associated with a database:

(i) Data graph: It is a labelled graph $D(V_D, E_D)$, where every node $v$ represents an object of the database and has a label $\lambda(v)$ and a set of keywords, associated with it. Each edge $e$ from $u$ to $v$ is labeled with its role $\lambda(e)$ and represents a relationship between $u$ and $v$. Example for a role could be “cites”, which is a relation between a paper and another paper that it cites.

(ii) Schema graph: It is a directed graph $G(V_G, E_G)$ that describes the structure of D.

(iii) Authority Transfer Schema graph, $G^A(V_G, E^A)$ reflects the authority flow through the edges of the graph and is obtained from the schema graph by inserting for each edge $e_G = (u,v)$ of $E_G$, two authority transfer edges - a forward edge $e_G^f = (u,v)$ with an authority transfer rate, $\alpha(e_G^f)$ and a backward edge $e_G^b = (v,u)$ with an authority transfer rate, $\alpha(e_G^b)$, in the authority transfer schema graph. The intuition for having two edges in the Authority Transfer Schema Graph, corresponding to each edge in the Schema Graph is that, the authority could flow in both directions at different rates.

(iv) Authority Transfer Data Graph, $D^A(V_D, E_D^A)$ is constructed from the data graph $D(V_D, E_D)$ and the authority transfer schema $G^A(V_G, E^A)$ by the following method: For every edge $e = (u, v) \in E_D$, add two edges $e^f = (u, v)$ with authority transfer rate $\alpha(e^f)$ and $e^b = (v, u)$ with authority transfer rate $\alpha(e^b)$. Let $e^f$ be of type $e_G^f$ and let $OutDeg(u, e_G^f)$ be the number of outgoing edges from u, of type $e_G^f$. Then, authority transfer rate $\alpha(e^f)$ is defined as:

$$\alpha(e^f) = \begin{cases} \frac{\alpha(e_G^f)}{OutDegree(u, e_G^f)} & if OutDegree(u, e_G^f) > 0 \\ 0 & if OutDegree(u, e_G^f) = 0 \end{cases}$$

The idea is that, for a keyword $k$ in the query, authority is initially present in the nodes that contain $k$. This authority, then flows to other nodes through the edges that connect them to the former ones, in correspondance with the authority transfer data graph. The final authority vested in a node is the score of that node with respect to $k$. Depending on the keywords and the query semantics, the scores of the nodes with respect to the individual keywords are combined as described in Section 4.2.2, to get the total score, which can be used to rank the nodes with respect to the query, and presented to the user.

**ObjectRank Index:** It is an inverted index, which stores for each keyword $w$, a list of $\langle id(u), r^w(u) \rangle$ pairs for each object $u$ in the data graph, that has $r^w(u) \geq threshold$, sorted by descending $r^w(u)$. Here, $threshold$ is an input parameter, which determines the minimum ObjectRank required, for an object to be entered in the index.

### 4.2.2 Random Surfer model for Ranking

The ranking method used in ObjectRank is in similar lines of the PageRank [BP98] used by the Google search engine for the ranked retrieval of web pages. Initially, a large number of random surfers start from objects containing the specified keyword and they traverse the database graph along the edges. At any point of time, a random surfer at a node either moves to an adjacent node by moving along an edge or jumps to a randomly chosen node containing the keyword. As time goes to infinity, the expected percentage of surfers at each node converges to a limiting value which is the ObjectRank of that node.

**Keyword-specific and Global ObjectRanks:** Each node $v_i \in V_D$ will be conferred with two ranks - Keyword-specific ObjectRank and Global ObjectRank. Keyword-specific ObjectRank of a node gives the importance of that node with respect to that keyword, while, the Global ObjectRank gives the general importance of that node, regardless of the query.

Let $w$ be a keyword in the query and let $S(w)$ be the set of objects that contain the keyword $w$, called the keyword base set. Then, the keyword-specific ObjectRank $r^w(v_i)$, of node $v_i$ is obtained by resolving the equation :

$$\mathbf{r}^w = d\mathbf{A}\mathbf{r}^w + \frac{(1-d)}{|S(w)|}\mathbf{s} \tag{4.1}$$

where $A_{ij} = \alpha(e)$ if there is an edge $e = (v_j, v_i)$ in $E_D^A$ and 0 otherwise. $\mathbf{s} = [s_1, ..., s_n]^T$ is the base set vector for $S(w)$ where, $s_i = 1$ if $v_i \in S(w)$ and 0 otherwise. $d$, called the damping

factor, controls the importance of the base set. The idea of damping factor too has been acquired from the PageRank used by web search engines. In this setting, it determines the amount of authority transferred by an object to its neighbours. A lower value of $d$ favors objects that actually contain the keywords over objects that acquired ObjectRank from their neighbours.

Global ObjectRank is calculated in a similar way using the equation 4.1, except that all nodes of the data graph are included in the base set. Thus, with probability $\frac{(1-d)}{|V_D|}$, the random surfer will jump into any one of the nodes in $V_D$.

***Multiple-Keyword Queries:*** Multiple-Keyword queries can be of two semantic types: AND and OR. These semantics can be explained by extending the random surfer model described in 4.2.2. Let the multiple-keyword query be $w_1, ..., w_m$. Consider $m$ independent random surfers, where the $i^{th}$ surfer starts from the keyword base set $S(w_i)$. At a given point of time, the probability that the m random surfers are simultaneously at node $v$ gives the ObjectRank of $v$ with respect to the m keywords, in the AND semantics. Whereas, the probability that atleast one of them is at node $v$ gives its ObjectRank under the OR semantics. Describing formally,

$$r_{AND}^{w_1,\ldots,w_m}(v) = \prod_{i=1,\ldots,m} r^{w_i}(v)$$
$$r_{OR}^{w_1,\ldots,w_m}(v) = \sum_{i=1,\ldots,m} r^{w_i}(v)$$

### 4.2.3 ObjectRank Index creation

For a general authority transfer data graph $D^A$, the ObjectRank of the nodes are calculated using the equation 4.1, by repeating it until convergence or until the difference between the $r(v)$ values obtained for consecutive iterations is lesser than *epsilon*, which is the convergence constant. This method is less efficient since it has to iterate until convergence and also because, it need to access $D^A$ many times during its execution.

But, in many applications, $D^A$ is a DAG, in which case, an improved algorithm can be employed. This algorithm makes only a single pass of $D^A$ and computes the precise solution of 4.1. The key step is to sort the nodes of $D^A$ topologically. Then solve for the equation 4.1, which can be done in a single iteration, for each keyword. The intuition behind this is that, ObjectRank is transferred only in the direction of the topological ordering. Topologically sorting a graph $G(V, E)$ takes $\theta(V + E)$ time in the general case. But, this can be brought down for databases where a temporal or some other ordering exists, by first sorting according to that ordering.

**Almost-DAG Algorithm :** A more practical situation is when the authority transfer data graph $D^A$ is such that, there is a set $U$ of backedges which when removed, makes $D^A$, a DAG. When the size of $U$ is small, when compared to the number of nodes in the entire graph, $D^A$ is said to be Almost-DAG. Identifying the minimum set of backnodes is NP-complete. However, in most cases, the semantics of the database can be of help in identifying the set of backedges. For example, a database where there is inherent temporal ordering between the objects, then, a backnode is an object referencing an object with a newer timestamp. In an Almost-DAG setting, each node can be assumed to possess two kinds of ObjectRanks:

(i) Backedges-ObjectRank: This rank is due to the backedges. To calculate the backedges-

ObjectRank, each backnode is assigned a value $c_i$, which represents its ObjectRank. Now, the form of their propogation to the rest of the nodes in $D^A$ is found out by assuming that the backedges are the only source of ObjectRank and then, making a pass of the DAG in the topological order. These values are stored in a matrix $C$. Hence, $C_{ij}$ specifies the quantity by which the $c_j$ should be multiplied, when calculating the ObjectRank of $i^{th}$ node.

(ii) DAG-ObjectRank: It is calculated from the DAG obtained by removing the backedges. But the degree of the nodes are maintained as before. In the calculation, $c_i$'s of the nodes are used to obtain a system of linear equations. This is subsequently solved to obtain the actual numerical values of $c_i$'s.

The total ObjectRank is then calculated as the sum of the Backedges-ObjectRank and the DAG-ObjectRank.

## 4.3 Proximity Search in Type-Annotated Corpora

The system given by Chakrabarti et. al. in the paper [CPD06], introduces a new class of text proximity queries to find an instance of a given `answer type` near `selector` tokens matching given literals or satisfying given ground predicates. The `answer types` or `atypes` can be person, place, distance etc. Such queries are useful in information extraction, question answering etc.

### 4.3.1 Data Model

The system requires an atype taxonomy, which is a DAG whose nodes are atypes and edges represent $is - a$ relation. Each document in the corpus is taken as a sequence of tokens. Some of these tokens will be connected to the nodes in the atype taxonomy, by an annotator module.

The system employes many kinds of indexes:

**Stem and full atype indexes:** A stem index maps stems to posting lists where, each posting is a record containing a document ID where the stem appears, the number of times it appears in the document, and a list of token offsets where it appears. While indexing a stem, all atypes attached to that token are also looked up in the atype taxonomy. All these atypes are also indexed as if they all occurred at the same offset as the token, in the full atype index. Both these indexes are inverted indexes and they can be built in the first pass over the documents. But, the full atype index could be almost the size of the uncompressed original corpus, which is unacceptable in most search applications. This can be avoided by choosing only a subset of atypes to index. The atype subset is selected based on an estimate of the probability of seeing those atypes in a new query.

**Reachability index:** Given two atypes $a1$ and $a2$, or an atype $a1$ and a token $w$, this index can tell if $a2$ or $w$ $is - a$ $a1$ in $O(1)$ time. i.e. whether $a1$ is a generalization of $a2$ or $w$ in the atype taxonomy. Hence, given a token, it can be easily verified if it is a candidate.

**Forward index:** This index can give the actual token, given $(docid, tokenOffset)$. This is useful in generating query-specific snippets along with the top responses. The forward index

is built in two passes over the corpus. In the first pass, the frequency of each token in the corpus is counted. Then the tokens are sorted by frequency and variable-length integer IDs are assigned to them - with shorter IDs to frequent tokens. In the second pass, the corpus is rescanned and the packed bit-vectors for each document are stored.

### 4.3.2 Query and Answer Model

A query consists of two parts - an *atype* from the taxonomy and a set of predicates on token strings. When the predicates are simple equality on stems or strings, the string literals are called *selectors*. Hence, proximity queries are of the form: *type=atype NEAR* $S_1 S_2 ... S_k$ which finds an instance of answer type *atype* near selector tokens $S_1 S_2 ... S_k$. Here, nearness is a function of the selectors, their frequency in the corpus, and their distance from the candidate answer. Any token in the corpus that is connected to a descendant of *atype* is a candidate answer token. A pruning policy adopted here, is to admit a candidate only if at least one selector appears within $W$ tokens of the candidate. $W$ is typically set to 50.

### 4.3.3 Scoring Model

The score of a candidate token depends on the matched selectors in the vicinity, the distance of the candidate from those selectors and the manner in which contributions from different selectors are aggregated at the candidate token.

**Selector energy:** Each selector $s$ possesses some $energy(s)$, which can spread into the candidates. A common notion of energy is the inverse document frequency (IDF) standard in IR, given by $N/N_s$ or its logarithmic form $log(1+N/Ns)$, where $N$ is the number of documents in the corpus and $N_s$ is the number of documents containing the selector token $s$.

**Gap and Decay:** $gap(w,s)$ denotes the gap between a candidate token $w$ and a matched selector $s$, which gives the number of tokens present between them. The amount of energy received by $w$ from $s$ is defined as $energy(s)decay(gap(w,s))$, where $decay(g)$ is a function of the gap. An important contribution of the paper, is a method for automatically learning this decay function. This has been described in Section 4.3.4.

**Score of a candidate:** Let $a$ be a candidate and $s$, a selector. $s$ can appear multiple times near $a$ and let this set be $s_i$. Then, score of $a$ is defined as :

$$score(a) = \oplus_s \oslash_i energy(s_i)decay(gap(s_i,a))$$

where $\oslash$ aggregates over multiple occurrences of $s$ and $\oplus$ aggregates over different selectors. Sum or max can be used as the aggregation functions, although sum behaves poorly as $\oslash$, because a low-IDF selector appearing often near a candidate can make the score less reliable.

### 4.3.4 Learning the Scoring Function

As mentioned before, the decay function, $\beta$, is automatically learned from the past query and answer logs. Let $\beta_j$ denote the decay at distance $1 \leq j \leq W$. Two methods for learning the decay have been proposed:

(i) Using Hinge Loss: Let $\mathbf{f}$ denote the feature vector. Then, the score assigned to it is given as $\mathbf{f} \; . \; \beta$ . Let $\mathbf{f}^{+}$ and $\mathbf{f}^{-}$ denote respectively, the feature vectors for the positive and negative contexts. Then, to learn $\beta$, create pairs of the form ($\mathbf{f}_i^{+}$,$\mathbf{f}_i^{-}$ ). Now, the constraints on $\beta$ can be written as, for all $i$, $\beta.(\mathbf{f_i^{+}} - \mathbf{f_i^{-}}) \geq 0$. This can be solved using the RankSVM method by solving the minimization problem: $min_{\beta,s\geq 0}(\frac{1}{2}\beta'\beta + C\sum_i s_i)$ such that, for all $i$, $\beta.\mathbf{x_i} + s_i \geq 1$. Here, $\mathbf{x_i}$ stands for $\mathbf{f_i^{+}} - \mathbf{f_i^{-}}$ and $C$ is a parameter that is tuned according as the complexity of the model required.

(ii) Using exponential loss: In the optimization target for RankSVM, $x_i$ is assigned a penalty of $C.max\{0, 1 - \beta.x_i\}$. An upper bound on this penalty is $C.exp(-\beta.xi)$. This gives the unconstrained optimization called RankExp: $min_\beta(\frac{1}{2}\beta'\beta + C\sum_i exp(-\beta.xi))$. This could be lesser accurate than the one learned by Hinge loss, but it scales up better, with the training set size.

The authors of the paper found that typically, $\beta$ vector is not monotonically decreasing with gap. To be specific, the results were such that, the maximum value of importance was not for matches at values near zero gap, but at values closer to gap a of 5. This is explained as follows: usually, selectors are named entities and are connected to the answer through articles, prepositions, etc., which creates gap. Hence, the $\beta$ vector gives allowance for these connecting words, which appear at gaps closer to zero.

# 5 Searching using Graph Patterns

The most popular form of keyword querying is by means of a list of terms or keywords. But this offers less scope for describing the constraints that the answer is required to satisfy and hence, is less expressive. A powerful method of querying is based on tree or graph patterns. A graph pattern, in general, consists of nodes which can be variables or constants, and edges which represent a constraint on the relationship between the nodes that it connects. The answer returned by the system must have valid data objects that can take the place of the variables and also, satisfy the edge constraints. Two systems that support graph/tree pattern queries are discussed in the sections below.

## 5.1 FleXPath

This section is intended to give an overview of FleXPath [AYLP04], which is a system used for querying eXtensible Markup Language (XML) data. XML data is organized in the form of a tree, by means of tags. This data will be entered into an XML database to enable querying. The process of converting XML data into appropriate representation and entering into the database, will not be discussed here.

### 5.1.1 Tree Pattern Query

The query model of FleXPath is called *tree pattern query* (TPQ). A TPQ is of the form $(T, F)$, where $T$ is a rooted tree with nodes denoting variables, and the edges denoting parent-child (pc) or ancestor-descendant (ad) relationships. $F$ is a predicate expression which specifies

constraints on the contents of the nodes. One of the nodes, usually, the root node, is designated as the answer to the query. The edges in $T$ represent the structural predicates which must be satisfied by the answer. Another kind of predicate, called value-based predicate, specifies constraints on type or contents of a node. For e.g., `$1.tag = article` specfies that the node denoted by `$1` must be of `article` type. `$2.age > 50` specifies that the value of the `age` attribute of `$2` must be greater than `50`. FleXPath defines another value-based predicate called `contains`, which can be expressed in the form `contains($i, exp)`. Here `exp` is an expression involving keywords, which can contain conjunctions, disjunctions etc. This predicate specifies that there must be a node in `$i` that satisfies the expression `exp`.

### 5.1.2 Answer to a Tree Pattern

An XML document $d$ is said to match a TPQ $(T, F)$, if the nodes of $d$ map to that of $T$ and also satisfy the predicate expression $F$, and the edges of $d$ satisfies the structural constraints given by the edges of $T$. The answer corresponding to every match $d$, is the node of $d$ that matches the distinguished node of the tree pattern query.

But, this semantics describes an exact match, which may be satisfied by only a handful of documents and hence defeats the purpose of keyword querying. FleXPath deals with this, by defining a relaxation to the original query, such that, more documents can be retrieved. A relaxation to a query expression admits, along with the documents that match the original one, some other documents as well. These extra documents may be such that they don't satisfy some structural or value-based predicates. A relaxation may be done by dropping some predicates from the query, introducing unions, promoting a `contains` predicate into the parent, etc. The details of relaxation techniques used and the algorithms for finding the matches, are not discussed in detail here. The answers thus obtained are assigned a relevance score. The ranking schemes used in FleXPath are described in the section below.

### 5.1.3 Ranking Scheme and Predicate Penalty

As mentioned in the previous section, FleXPath relaxes a query to admit more potential candidates for the query. In general, a good ranking scheme should be such that, the answers that are obtained for the relaxed query must be ranked lower than that of the original query. This property is referred to as ***Relevance Scoring***. Another desirable property of the ranking scheme is ***Order Invariance***, which requires that the rank/score given to an answer to the relaxed query must be independent of the order in which the predicates were dropped from the original query, to obtain the relaxed one.

An important notion in the ranking scheme is that of ***Predicate Penalty*** , which measures the extend of the loss of context, when a predicate is dropped to get the relaxed query. For this, each predicate $p$ in the query $Q$ is given a certain weight, specified as $w_Q(p)$, which is a measure of its importance in the query. Suppose $pc(\$i, \$j)$ is a parent-child relationship(edge) specified in the query. So, dropping this constraint from the query will be equivalent to having just the ancestor-descendant constraint $ad(\$i, \$j)$ in the query. Let $\#_{rel}(t_i, t_j)$ denote the number of pairs of nodes of type $(t_i, t_j)$, that are connected by the relation $rel$. Then, the penalty of the

above relaxation is defined as:

$$penaltyOfDropping(pc(\$i,\$j)) = \frac{\#_{pc}(t_i,t_j)}{\#_{ad}(t_i,t_j)} w_Q(pc(\$i,\$j))$$

The intuition behind this definition of penalty is that, if a large number of nodes satisfying $ad(\$i,\$j)$ also satisfy $pc(\$i,\$j)$, then the loss of context in relaxing the $pc$ constraint is high. The penalty also depends on the weight attached to the predicate.

In similar lines, the penalty of dropping an ancestor-descendant predicate and `contains` predicate are given by:

$$penaltyOfDropping(ad(\$i,\$j)) = \frac{\#_{ad}(t_i,t_j)}{\#(t_i) \times \#(t_j)} w_Q(ad(\$i,\$j))$$

$$penaltyOfDropping(contains(\$i,exp)) = \frac{\#_{contains}(\$i,exp)}{\#_{contains}(\$l,exp)} w_Q(contains(\$i,exp))$$

where, $\#(t)$ is the number of elements of tag $t$ in the document and $\$l$ is the parent node of $\$i$ in $Q$ ( and hence, dropping the `contains($i, exp)` corresponds to promoting it to the parent, $\$l$).

Score of an answer has two parts - structural score denoted by $ss$ and keyword score denoted by $ks$. Let $Q$ be the original query and let $P$ be the set of all predicates in it. Let $S$ denote the set of predicates that have been dropped from $P$ to obtain a relaxed version of $Q$. Then structural score is given by:

$$ss = \sum_{p \in P} w_Q(p) - \sum_{p \in S} \pi(p)$$

where, $\pi(p)$ is the penalty incurred for dropping predicate p, as described above. The keyword score is defined as the the weighted sum of the scores assigned to it by an IR engine. The final score can be an ordered pair or just a number. The ordered pairs can be of two kinds:

***Structure first:*** score is given by $(ss, ks)$.

***Keyword first:*** score is given by $(ks, ss)$.

In this case, the answers are ranked by ordering them lexicographically.

The numerical score is given by an arithmetic function that combines $ks$ and $ss$, like the sum of the two scores. For this, the ranking is done by just sorting them on the decreasing values of their scores.

## 5.2 The NAGA system

The NAGA system [KSI+08] is yet another system that supports graph pattern queries. NAGA stands for the infinite amounts of knowledge that remains coiled up in World Wide Web. In NAGA, the basic units are not words, but facts and fact templates. Facts are binary relationships derived from the Web. This is described in detail in Section 5.2.1.

### 5.2.1 Data Model

NAGA represents data as a graph, in which nodes are labeled with entities and edges, with relationships. Representing formally, the data model is a directed, weighted, labeled multi-graph

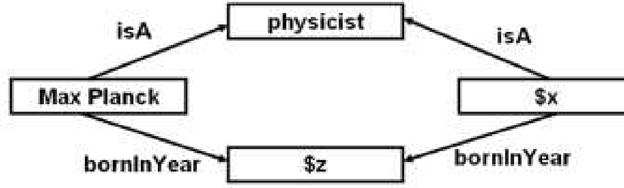



Figure 1: Discovery query example

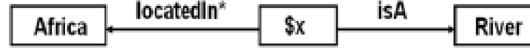



Figure 2: Regular expression query example

$G = (V, E, L_V, L_E)$. $V$ is a set of nodes, $E$ is a multi-set of edges, $L_V$ is a set of node labels and $L_E$ is a set of edge labels. Each node $v$ is uniquely assigned a label $l(v) \in L_V$ and each edge $e$, a label $l(e) \in L_E$. Each edge, together with its end nodes, represents a fact. Each fact has a confidence value associated with it, which depends on the estimated accuracy $acc(f, p)$ with which the fact $f$ was extracted from a page $p$ and the trust $tr(p)$ in $p$. The accuracy value is dependent on the extraction mechanism and the trust in a page is computed by an algorithm similar to PageRank. The pages from which $f$ has been extracted are called witnesses of $f$. If $f$ was extracted from the witnesses $p_1, ..., p_n$, its confidence value is given as:

$$c(f) = \frac{1}{n} \sum_{i=1}^{n} acc(f, p_i) tr(p_i) \tag{5.1}$$

Each fact is stored only once and hence the knowledge graph is free from redundancy.

### 5.2.2 Query and Language Model

The queries supported by NAGA are:

(i) Discovery query: to discover pieces of information. It is specified using a connected subgraph, whose nodes and edges may not be labeled. An example is given in Figure 1, which seeks physicists who were born in the same year as Max Planck.

(ii) Regular expression query : Specified by a subgraph in which atleast one of the edges is a regular expression. It helps to find out some particular path connecting pieces of information. For example, to find out the rivers located in Africa, a query of the form given in Figure 2, can be stated.

(iii) Relatedness query : This can be used to find out a broad relationship between pieces of information. Here, the subgraph has atleast one of the edges labeled with ‘connect’. For example, the question “How are Margaret Thatcher and Indira Gandhi related?”, can be stated as in Figure 3.

Figure 3: Relatedness query example

A statistical language model assigns a probability to a sequence of words, on the basis of a probability distribution. $P(g|q)$, which is the probability that the given query $q = q_1q_2...q_n$ was indeed generated by the subgraph $g$ , can be estimated as:

$$P(g|q) \sim P(q|g)P(g) \tag{5.2}$$

$$where, P(q|g) = \prod_{i=1}^{n} P(q_i|g) \tag{5.3}$$

Equation 5.3 assumes that the fact templates of the query are independent. This is based on the intuition that the facts are extracted independently. $P(g)$ is the prior belief that $g$ is relevant to any query. $P(q_i|g)$ is modeled as:

$$P(q_i|g) = \alpha\tilde{P}(q_i|g) + (1-\alpha)\tilde{P}(q_i)$$

which is a mixture of probability of drawing $q_i$ from $g$ randomly and drawing $q_i$ randomly from the entire knowledge graph, respectively. Here, $0 \leq \alpha \leq 1$ is either automatically learned or set to an emphirically calculated value.

The confidence and informativeness for query $q_i$ is defined as:

$$P_{conf}(q_i|g) = \prod_{f \in match(q_i,g)} P(fholds) \tag{5.4}$$

$$P_{info}(q_i|g) = \prod_{f \in match(q_i,g)} P_{finfo}(f|q_i) \tag{5.5}$$

where $P(fholds)$ is the confidence, $c(f)$ given by Equation 5.1 and $P_{finfo}(f)$ is the informativeness of the fact $f$, which is a function of its number of witnesses.

Now, $\tilde{P}(q_i|g)$ is defined as a mixture of confidence and informativeness:

$$\tilde{P}(q_i|g) = \beta P_{conf}(q_i|g) + (1-\beta)P_{info}(q_i|g)$$

where $0 \leq \beta \leq 1$ is calibrated emphirically.

### 5.2.3 Answer Model

Answer to a query $Q$ is a subgraph $A$ of the knowledge graph that matches $Q$. The edges of the subgraph $A$ is also marked with the confidence of the relation. Different answers that match a query are ranked using the $P(A|q)$ calculated using the Equation 5.2.

The scoring model captures the following:

(i) Confidence: It expresses the certainity about a specific fact and is independent of the query and the popularity of the fact. The confidence in a fact depends on the trust in

pages from which the fact has been extracted and the accuracy with which it has been extracted. Hence, facts extracted from authoritative pages, with high accuracy, will be given a higher score, by the scoring model. Confidence is calculated using the Equation 5.4, discussed in the previous section.

(ii) Informativeness: It captures how relevant is a fact for a given query. This is dependent on the formulation of the query. A fact is deemed to be relevant if it is highly visible in the web. This is based on the intuition that the more the number of pages that state the fact, the higher is the likelihood that the fact is true and is an importance piece of information. Informativeness is calculated using the Equation 5.5.

(iii) Compactness of the resulting graph: Compactness is important because a large graph connecting the nodes may not make much sense. Intuitively, if the number of facts required to connect the nodes is high, then, the logical relationship conveyed by the graph is more likely to be absurd. In NAGA, the compactness of the answers is implicitly captured by their likelihood given the query (calculated using Equation 5.2), because the likelihood of an answer graph is the product over the probabilities of its component facts. Hence, the likelihood and thus, the compactness decreases as the number of facts increases.

# 6 Conclusion and Future Work

In this report, different models of keyword search were discussed, giving special attention to the systems that implement search on databases. Different query models like sequence of words and graph patterns were explained. Various techniques employed for finding answers based on their relevence were also looked into. Different scoring models from simple ones based on the size of the answer, to complex ones that used notion of prestige and proximity were also discussed. The importance of efficiency in terms of space and time, were also brought into notice. A concluding remark is that, each of the systems discussed in this report were unique in some sense, and hence, none of them can be regarded as a clear winner over the others. Hence, if a new system is to be built, it would be excellent if some of the important aspects of each of them (or an improved version) could be included.

Future work will be oriented towards bringing in, more semantics to the answers generated. The database contains enough amount of information to facilitate this; but the direction and techniques to be used requires more research. Another important aspect of query answering is to provide background ideas and the connection between these and the central theme. An important work in this direction is described in [KSI06]. In the proposed system, the queries are free-form and they generate an entire multi-relation database, which is a logical subset of existing one. The idea is to provide the user with data that is implicitly related to the query, along with the items that are directly related, thus giving a proper insight into the original data. Apart from this, a keyword query system should allow for more expressive querying methods and produce more intuitive answers, presented in a better manner, for the user to grasp easily, the information conveyed.

## References


[ACD02] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. *ICDE*, 2002.

[AYLP04] Sihem Amer-Yahia, Laks V.S. Lakshmanan, and Shashank Pandit. FleXPath: Flexible Structure and FullText Querying for XML. *SIGMOD*, 2004.

[BHN+02] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. *ICDE*, 2002.

[BHP04] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. ObjectRank: Authority-Based Keyword Search in Databases. *VLDB Conference*, 2004.

[BP98] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *WWW Conference*, 1998.

[CPD06] Soumen Chakrabarti, Kriti Puniyani, and Sujatha Das. Optimizing Scoring Functions and Indexes for Proximity Search in Type-annotated Corpora. *DBLP Conference*, pages 717–726, 2006.

[GSVGM98] Roy Goldman, Narayanan Shivakumar, Suresh Venkatasubramanian, and Hector Garcia-Molina. Proximity Search in Databases. *VLDB Conference*, 1998.

[HGP03] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. *VLDB Conference*, 2003.

[HP02] Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. *VLDB Conference*, 2002.

[KPC+05] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional Expansion For Keyword Search on Graph Databases. *VLDB Conference*, 2005.

[KSI06] Georgia Koutrika, Alkis Simitsis, and Yannis Ioannidis. Précis: The Essence of a Query Answer. *ICDE*, 2006.

[KSI+08] Gjergji Kasneci, Fabian M. Suchanek, Georgiana Ifrim, Maya Ramanath, and Gerhard Weikum. NAGA: Searching and Ranking Knowledge. *ICDE*, 2008.