NUITS: A Novel User Interface for Efficient Keyword Search over Databases

Shan Wang, Zhaohui Peng, Jun Zhang
Lu Qin, Sheng Wang
School of Information, Renmin Univ. of China
Key Laboratory of Data Engineering and
Knowledge Engineering, MOE of China
Beijing, China
{swang,pengch,zhangjun11,
qinlu,wangsheng1}@ruc.edu.cn

Jeffrey Xu Yu, Bolin Ding
Department of Systems Engineering and
Engineering Management
The Chinese Univ. of Hong Kong
Hong Kong, China
{yu,blding}@se.cuhk.edu.hk

ABSTRACT

The integration of database and information retrieval techniques provides users with a wide range of high quality services. We present a prototype system, called *NUITS*, for efficiently processing keyword queries on top of a relational database. Our *NUITS* allows users to issue simple keyword queries as well as advanced keyword queries with conditions. The efficiency of keyword query processing and the user-friendly result display will also be addressed in this paper.

1. INTRODUCTION

Keyword search over relational databases enables end users, who do not understand the underneath schema and *SQL*, to find the connections among the tuples stored in relations, with a given set of keywords. Here, the connections among the tuples are specified by foreign-key references. The recent studies can be categorized into two types according to the search mechanism they adopted, namely, *schema-graph-based* and *data-graph-based*. The former includes *DBXplore* [1] and *IR-Style* [3], whereas the latter includes *BANKS* [2, 4].

Consider a `DBLP` database [5] for citations among research papers written by authors using 4 relations: `Author`, `Paper`, `Write`, and `Cite`. The `Author` relation is with an author-id (`AID`) and an author name (`Name`). The `Paper` relation is with a paper-id (`PID`) and a title (`Title`). The `Write` relation specifies the relationship between a paper and an author using paper-id and author-id. `PID` and `AID` in the relation `Write` are foreign key references to `PID` and `AID` in the `Paper` relation and the `Author` relation, respectively. The `Cite` relation specifies the citation between two papers, and both attributes, `cite` and `cited`, are foreign key references to `PID` in the relation `Paper`. Here, a simple keyword query may contain several keywords, for example, `Codd` and `XML`. The results, for this 2 keyword query, include the papers that have `XML` in the title and cite a paper written by `Codd` with a possible sequence of citation relationships.

In this demonstration, we present a data-graph-based system called *NUITS* over a relational database system. In addition to supporting advanced keyword queries, *NUITS* achieves high efficiency for keyword query processing and provides users with different view mechanisms to assist them to view the complicated structure among tuple connections, which are considered as difficult tasks [8].

2. KEYWORD QUERY SPECIFICATION

NUITS supports several advanced keyword queries as well as simple keyword queries. An example of simple keyword query is given above. We discuss keyword specification followed by the discussions on advanced keyword queries.

- **Simple keyword**: A simple keyword is just a keyword, for example `database`.
- **Typed keyword**: Users do not need to know the underneath relational database schema when they issue keyword queries. But, because a keyword may appear in any attributes and in any relations, the results may be large and include many users do not need. In order to restrict the search space, typed keywords are introduced in *NUITS* which allows users to specify a keyword with a type. Here a type can be either relation-name or attribute-name. For example, `Paper:database` means that a keyword of `database` appearing in the `Paper` relation. In addition, we introduce a wildcard `*` for any possible keyword. For instance, if a user is interested in any authors who wrote a paper on `database`, he/she can issue a 2 keyword query with `Author:*` and `database`. Since casual users may not know the exact relation or attribute name, *NUITS* supports aliases. The same query in the above example can also be written as `Writer:*` and `database`, as long as the alias "`Writer`" has been configured in advance by system administrators.
- **Conditional keyword**: *NUITS* allows users to specify conditions associated with a keyword. For exam-

VLDB '06, September 12-15, 2006, Seoul, Korea.

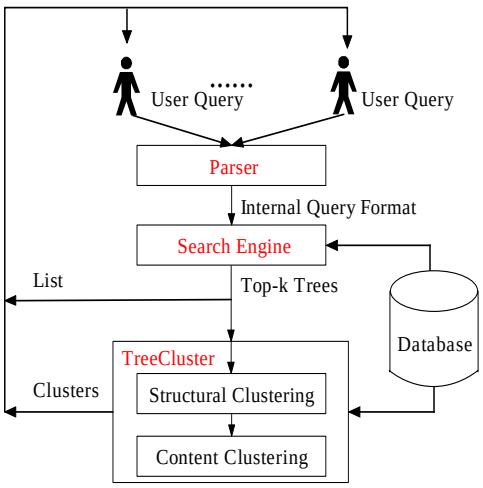

Figure 1: The *NUITS* architecture

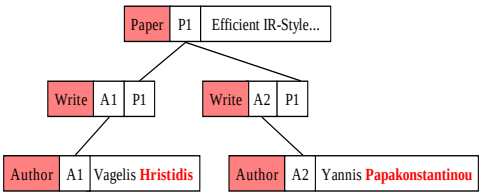

Figure 2: An example of tuple-connection-tree

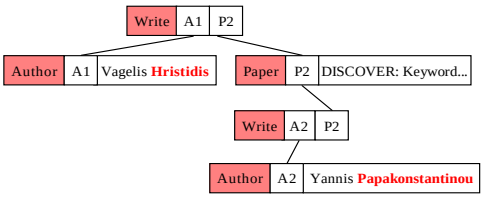

Figure 3: Another example of tuple-connection-tree

ple, `database year>2000` specifies a condition associated with the keyword `database`. The condition means that, if a tuple containing the keyword `database` has an attribute called `year`, its value must be greater than `2000`. Instead of $>$, the other comparators such as $<$, $\leq$, $=$, $\geq$ and $\neq$ can also be used. Note: a keyword can be associated with multiple conditions. In addition, *NUITS* provides a special operator ~ for approximation keyword. For example, `database year~2000` means that the tuple-connection-trees with nodes (tuples) containing a numerical value of `year`, which is closer to year 2000, will be given a smaller cost.

A keyword query is a set of keywords associated with Boolean operators such as `AND`, `OR` and `NOT`. The keyword query specification is given below.

$Q ::= p \mid (Q) \mid Q$ `AND` $Q \mid Q$ `OR` $Q \mid$ `NOT` Q

Here, p means a keyword which can be a simple keyword, typed keyword, or conditional keyword. Q means a keyword query. For instance, a keyword query, `Ullman AND (database OR algorithm)`, is to retrieve all the papers on `database` or `algorithm` written or cited by `Ullman`.

3. ARCHITECTURE

The architecture of *NUITS* is shown in Figure 1. The system adopts Browser/Server architecture. The browser allows users to issue keyword queries and view the resulting connections among tuples in a relational database. The server consists of three modules: a parser module, a search engine module, and a display module (denoted *TreeCluster*). Keyword queries are first parsed by the parser, which transforms various queries (simple queries and advanced queries) into an internal format. The search engine processes the internal query format in two steps. First, it identifies all the tuples in *RDBMS* that contain at least a keyword using a full-text index built on top of the *RDBMS*. Second, it finds the top-k resulting tuple-connection-trees, based on a weighting scheme. The resulting top-k tuple-connection-trees can be listed directly or further clustered into clusters according to each user's preference by the *TreeCluster* module.

In the following, we mainly discuss the search algorithm and the tree clustering.

4. SEARCH ALGORITHM

The search algorithm used in the search engine is outlined below. Like *BANKS* [2, 4], we model a relational database as a weighted graph, $G(V, E)$. Here V is a set of nodes representing tuples and E is a set of edges representing foreign-key references among tuples. An edge, $(u, v) \in E$, represents a foreign key reference between two nodes (tuples), u and v, if u has a foreign key matching the primary key attributes of v, or v has a foreign key matching the primary key attributes of u. The weights on nodes and edges are preassigned [2, 4].

Given a l keyword query, $p_1, p_2, \cdots, p_l$, against a relational database or equivalently the corresponding graph $G(V, E)$. Let $V_i \subseteq V$ be a set of nodes that contain the keyword p_i. An answer (tuple-connection-tree) to such a keyword query is a weighted and connected tree containing at least one node from each V_i. The problem we are targeting is how to find top-k minimum cost tuple-connection-trees. In order to improve the algorithm given in [2, 4], we proposed a dynamic programming approach to find the optimal top-1 with the time complexity of $O(3^L \cdot N + 2^L((L + \log N) \cdot N + M))$, where N and M are the numbers of nodes and edges in the graph G. Because the number of keywords, L, is small in keyword queries, this solution can handle graphs with a large number of nodes efficiently. It is important to note that our solution can be easily extended to support top-k. That is, we compute top-k minimum cost tuple-connection-trees one-by-one incrementally, and do not need to compute or sort all tuple-connection-trees in order to find the top-k results.

5. TREE CLUSTERING

The search engine will report the top-k minimal cost tuple-connection-trees. However, a potential problem is how to select such a parameter k. When k is small, a user may not be able to find the expected tuple-connection-trees. When k is large, a user may find it difficult because there are too

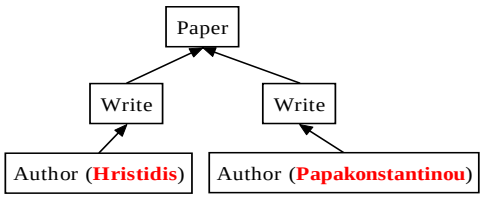

Figure 4: The structural pattern of the two trees in Figure 2 and Figure 3

many such trees. In order to assist users to find the needed tuple-connection-trees, in *NUITS*, we propose to cluster the similar trees into clusters. Two trees, t_i and t_j, are in the same cluster, if t_i and t_j are isomorphic to each other. Here, we consider trees as labeled trees at the schema level. Then, t_i and t_j are isomorphic to each other, if there is a one-to-one mapping from nodes of t_i to nodes of t_j. The details will be discussed below using examples.

Consider a keyword query with two keywords, `Hristidis` and `Papakonstantinou`. The keyword query intends to find out the papers co-authored by the two authors. We show two tuple-connection-trees, t_1 and t_2, in Figure 2 and Figure 3, respectively. Here, t_1 represents a joint work on "Efficient IR-Style Keyword Search over Relational Databases", and t_2 represents a joint work on "DISCOVER: Keyword Search in Relational Databases". The two trees, t_1 and t_2, look different because the resulting tuple-connection-trees are not rooted trees. But the two trees, t_1 and t_2, are tree isomorphic, if we ignore the details and concentrate on their structure by labeling nodes/edges of t_1/t_2 using the relation name where they belong as node-labels and foreign-key references as edge-labels. The structural pattern of the two trees are given in Figure 4 where the edge labels are omitted.

The details of labeling nodes/edges in tuple-connection-trees are given below. First, there are two kinds of nodes in the resulting tuple-connection-trees, namely, the nodes that contain one or many keywords, and the nodes that do not contain any keyword but are used to connect those nodes with keywords. We call the former and latter as K-node and C-node, respectively. For C-nodes, we label them using the relation-name where they belong. For K-nodes, because they may contain several keywords and a keyword may appear in several attributes of a node (tuple), we concatenate its relation-name, the attribute-name containing keywords and the keywords themselves as the label of K-nodes. For edges, we concatenate the primary-foreign key attributes as their labels. Note: tree isomorphic is performed on the labeled trees discussed above.

Next, we discuss how to select a represented tree for a cluster. Consider the two tuple-connection-trees, t_1 and t_2 shown in Figure 2 and Figure 3. Both can be selected as a representative tuple-connection-tree for the cluster. We solve this problem by determining the root node of trees that are isomorphic where the root node contains as much information as possible. Several rules are discussed in [6]. The rules include choosing a node with maximum degree or closing to the tree center.

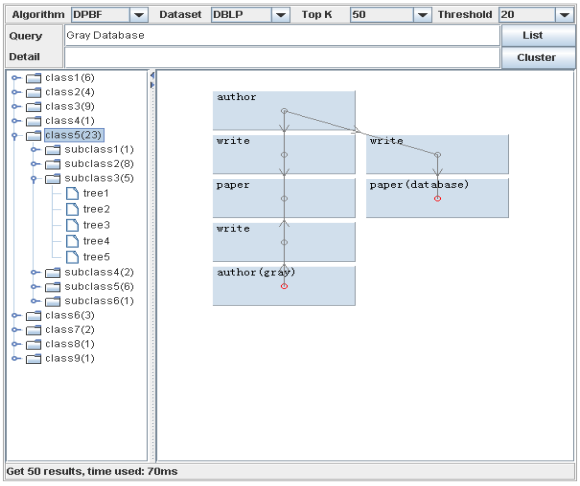

Figure 5: The structural-level clustering

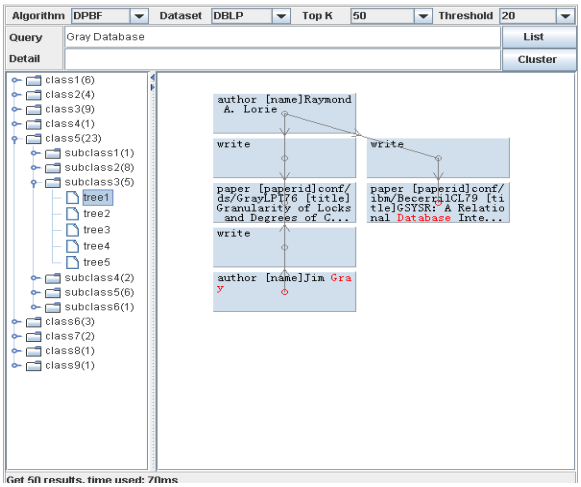

Figure 6: The tuple-connection-tree

The tree clustering helps users to see the general structure of tuple-connection-trees. The details of tree clustering, denoted *TreeCluster* is reported in [6]. *TreeCluster* can cluster tuple-connection-trees in two levels, at the structural level and the content level.

- **Structural-Level Clustering**: The structural-level clustering is to cluster trees using the tree isomorphic mentioned above. Consider a 2 keyword query against `DBLP` database, with `Jim Gray` and `Database`. In the resulting tuple-connection-trees, there are several potential clusters. For example, `Jim Gray` writes papers about `Database`; `Jim Gray`'s papers are cited by papers about `Database`; and `Jim Gray`'s papers cite other papers about `Database`. All the resulting tuple-connection-trees can be clustered into several clusters.
- **Content-Level Clustering**: The content-level clustering further clusters tuple-connection-trees if the size of the cluster is larger than a user given threshold, after structural-level clustering. The content-level clustering is based on keyword frequencies and content

similarity. For instance, consider a 2 keyword query with `Gray` and `Database` against the `DBLP` database. The keyword `Gray` may appear in a small number of tuples in the `Author` relation, whereas `Database` appears in many tuples in the `Paper` relation. A user can cluster tuple-connection-trees into a smaller number of clusters based on the keyword frequencies. In this example, a user can cluster it based on content of author names, because it occurs less frequently, and can result in a smaller number of clusters. There may be clusters of tuple-connection-trees for different authors. For example, a cluster of tuple-connection-trees for `Jim Gray` and a cluster of tuple-connection-trees for `W.A. Gray`.

The structural-level clustering is based on isomorphism of labeled trees, and the content-level clustering is based on content of tuples. So the two levels of clustering are cost effective, having slight effect on original system efficiency as experiments demonstrate.

6. INTERFACE OVERVIEW

The current design of the graphical user interface of *NUITS* is shown in Figure 6. It consists of several components. We discuss them below.

- **Keyword Query Input**: This component is for users to type in keyword queries.
- **Database**: There are two databases in this demonstration, `DBLP` [5] and `MDB` [7]. `MDB` consists of data of a movie recommendation system, including 3 relations: `User`, `Movie` and `Evaluate`. The `User` relation is with a user-id (`UID`) and a user name (`Name`). The `Movie` relation is with a movie-id (`MID`) and a title (`Title`). The `Evaluate` relation specifies the rating a user gives to a movie using user-id and movie-id. `UID` and `MID` in the relation `Evaluate` are foreign key references to `UID` and `MID` in the `User` relation and the `Movie` relation, respectively.
- **Algorithm**: We support three algorithms `BANKS` [2], `B-BANKS` [4], and our dynamic programming solution.
- **Input-Parameter Selection**: Users can select k for finding top-k minimum cost tuple-connection-trees, and select the *threshold* to cluster tuple-connection-trees in the content-level clustering.
- **Out-Parameter Selection**: Users can select how to view the resulting tuple-connection-trees. They can review the results either in the format of list of tuple-connection-trees or in the format of clusters of tuple-connection-trees, by selecting either `List` or `Cluster` in the interface of *NUITS*.
- **Tree List**: The resulting tuple-connection-trees are listed on the left side of the interface.
- **Two-Level Clustering**: A user can view the common structure of a cluster of tuple-connection-trees at the structural level, if he/she clicks the cluster of tuple-connection-trees. The common structure of the clusters of tuple-connection-trees is shown in the main window of the interface. Under the structural-level clusters, there are possible content-level clusters. A user can further view the clusters of tuple-connection-trees at the content-level. Finally, a user can view each individual tuple-connection-tree in the two levels of clusters.

  Figure 5 shows the result for the keyword query with `Gray` and `Database` against the `DBLP` database, when the parameter k is 50. In Figure 5, it shows the display when the user views one out of 9 clusters of tuple-connection-trees at the structural-level. Because the number of tuple-connection-trees in the 5th cluster at structural-level is 23, which is greater than the user-given threshold 20, the 5th cluster is further clustered into 6 clusters at the content-level. Figure 6 shows the same example when a user views an individual tuple-connection-tree.
- **Detailed Information**: The detailed information of a tuple can be displayed in the `Detail` field after it is clicked.

We also provide an interface for system administrators to manage and configure *NUITS*, including system configuration for performance tuning and indexing maintenance, and personalized user interface design.

7. ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No.60473069 and No.60496325).

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A System for Keyword-Based Search over Relational Databases. In *Proc. of ICDE'02*, 2002.

[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *Proc. of ICDE'02*, 2002.

[3] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In *Proc. of VLDB'03*, 2003.

[4] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional Expansion for Keyword Search on Graph Databases. In *Proc. of VLDB'05*, 2005.

[5] M. Ley. DBLP: Computer Science Bibliography. `http://dblp.uni-trier.de/xml/`.

[6] Z. Peng, J. Zhang, S. Wang, and L. Qin. Treecluster: Clustering Results of Keyword Search over Databases. In *Proc. of WAIM'06*, 2006.

[7] J. Riedl and J. Konstan. MoveLens. `http://www.cs.umn.edu/Research/GroupLens`.

[8] S. Wang and K.-L. Zhang. Searching Databases with Keywords. *J. Comput. Sci. Tech.*, 20(1):55–62, January 2005.