Keyword Searching and Browsing in Databases using BANKS

Gaurav Bhalotia* **Arvind Hulgeri**† **Charuta Nakhe**‡ **Soumen Chakrabarti**
S. Sudarshan§
Computer Science and Engg. Dept., I.I.T. Bombay
bhalotia@cs.berkeley.edu, charuta@pspl.co.in,
{aru,soumen,sudarsha}@cse.iitb.ac.in

Abstract

With the growth of the Web, there has been a rapid increase in the number of users who need to access online databases without having a detailed knowledge of the schema or of query languages; even relatively simple query languages designed for non-experts are too complicated for them. We describe BANKS, a system which enables keyword-based search on relational databases, together with data and schema browsing. BANKS enables users to extract information in a simple manner without any knowledge of the schema or any need for writing complex queries. A user can get information by typing a few keywords, following hyperlinks, and interacting with controls on the displayed results.

BANKS models tuples as nodes in a graph, connected by links induced by foreign key and other relationships. Answers to a query are modeled as rooted trees connecting tuples that match individual keywords in the query. Answers are ranked using a notion of proximity coupled with a notion of prestige of nodes based on inlinks, similar to techniques developed for Web search. We present an efficient heuristic algorithm for finding and ranking query results.

1. Introduction

Relational databases are commonly searched using structured query languages. The user needs to know the data schema to be able to ask suitable queries. Search engines on the Web have popularized an alternative unstructured querying and browsing paradigm that is simple and user-friendly. Users type in keywords and follow hyperlinks to navigate from one document to the other. No knowledge of schema is needed.

With the growth of the World Wide Web, there has been a rapid increase in the number of users who need to access online databases without having a detailed knowledge of schema or query languages; even relatively simple query languages designed for non-experts are too complicated for such users. Query languages for semi-structured/XML data are even more complex, increasing the impedance mismatch further.

Unfortunately, keyword search techniques used for locating information from collections of (Web) documents cannot be used on data stored in databases. In relational databases, information needed to answer a keyword query is often split across the tables/tuples, due to normalization. As an example consider a bibliographic database shown in Figure 1. This database contains paper titles, their authors and citations extracted from the DBLP repository. The schema is shown in Figure 1(A). Figure 1(B) shows a fragment of the DBLP database. It depicts partial information—paper title and authors—about a particular paper. As we can see, the information is distributed across seven tuples related through foreign key references. A user looking for this paper may use queries like "sunita temporal" or "soumen sunita". In keyword based search, we need to identify tuples containing the keywords and ascertain their proximity through links.

Answers to keyword queries on the Web are often only the starting point for further browsing to locate required information. Similar browsing facilities are needed in the context of searching for information from databases.

In this paper, we describe techniques for keyword searching and browsing on databases that we have developed as part of the BANKS system (BANKS is an acronym for Browsing ANd Keyword Searching). The BANKS system enables data and schema browsing together with keyword-based search for relational databases. BANKS enables a user to get information by typing a few keywords, following hyperlinks, and interacting with controls on the displayed results; absolutely no query language or program-

*Current affiliation: University of California, Berkeley
†Supported by an Infosys Fellowship
‡Current affiliation: Persistent Systems Pvt. Ltd., Pune, India
§Partly supported by an IBM Faculty Partnership Grant

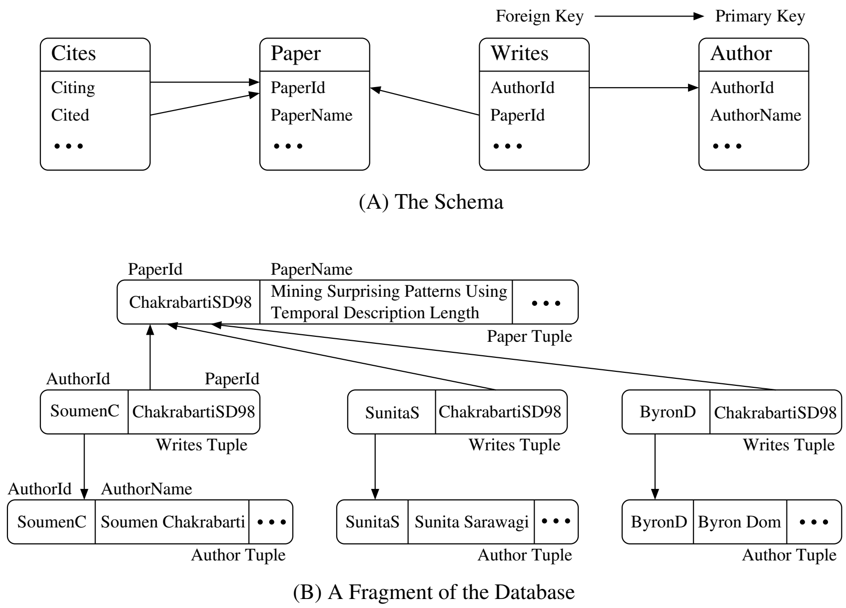

Figure 1. The DBLP Bibliography Databases

ming is required.

The contributions of this paper are as follows:

1. We outline a framework for keyword querying of relational databases. Our framework makes joins implicit and transparent, and incorporates notions of proximity and prestige when ranking answers.

   There has been a fair amount of earlier work on keyword querying of databases, including [6, 7, 12, 13]. We describe the connections of the BANKS model to related work on keyword search in Section 6.

2. We present novel, efficient heuristic algorithms for executing keyword queries.

3. We describe key features of the BANKS system.

Keyword searching in BANKS is done using proximity based ranking, based on foreign key links and other types of links. We model the database as a graph, with the tuples as nodes and cross references between them as edges. BANKS allows query keywords to match data (tokens appearing in any textual attribute), and meta data (e.g., column or relation name).

The greatest value of BANKS lies in near zero-effort Web publishing of relational data which would otherwise remain invisible to the Web [2]. BANKS may be used to publish organizational data, bibliographic data, and electronic catalogs. Search facilities for such applications can be hand crafted: many Web sites provide forms to carry out limited types of queries on their backend databases. For example, a university Web site may provide form interface to search for faculty and students. Searching for departments would require yet another form, as would searching for courses offered. Creating an interface for each such task is laborious, and is also confusing to users since they must first expend effort finding which form to use.

An approach taken in some cases is to export data from the database to Web pages, and then provide text search on Web documents. This approach results in duplication of data, with resultant problems of keeping the versions up-to-date, in addition to space and time overheads. Further, with highly connected data, it is not feasible to export every possible combination. For instance, a bibliographic database can export details of each paper as a Web document, but a query that requires finding a citation link between two papers would not be supported.

BANKS provides a rich interface to browse data, and automatically generates hyperlinks, corresponding to foreign keys and other links, on displayed results. BANKS helps create hierarchical and graphical views of data with hyperlink facilities built in. The BANKS system is developed in Java using servlets and JDBC, and can be run on any schema without any programming. BANKS is accessible over the Web at the URL: `http://www.cse.iitb.ac.in/banks/`

The rest of the paper is organized as follows: Section 2 outlines our graph model for representing connectivity information, as well as our model for answer relevance. Section 3 outlines an algorithm for incrementally finding the best answers to keyword queries. We present an overview of the browsing features of BANKS in Section 4. Section 5 outlines a preliminary evaluation of our system. We discuss related work in Section 6. Section 7 outlines directions for future work and Section 8 concludes the paper.

2. Database and Query Model

In this section we describe how a relational database is modeled as a graph in the BANKS system. First we evaluate various options available and describe our model informally, and then formalize it.

2.1. Informal Model Description

We model the database as a directed graph and each tuple in the database as a node in the graph. Each foreign-key–primary-key link is modeled as a directed edge between the corresponding tuples. This can be easily extended to other type of connections; for example, we can extend the model to include edges corresponding to inclusion dependencies, where the values in the referencing column of the referencing table are contained in the referred column of the referred table but the referred column need not be a key of the referred table.

Intuitively, an answer to a query should be a subgraph connecting nodes matching the keywords. Just by looking at a subgraph it is not apparent as to what information it conveys. We wish to identify a node in the graph as a central node that connects all the keyword nodes, and strongly reflects the relationship amongst them. We therefore consider an answer to be a rooted directed tree containing a directed path from the root to each keyword node. (The motivation for directionality is outlined later in this section.) We call the root node an *information node* and the tree a *connection tree*. Conceptually this model is similar to the one described in [13] although there are several differences which are detailed in Section 6.

In general, the importance of a link depends upon the type of the link i.e. what relations it connects and on its semantics; for example, in the bibliographic database, the link between the *Paper* table and the *Writes* table is seen as a stronger link than the link between the *Paper* table and the *Cites* table. The link between *Paper* and *Cites* tables would have a higher weight. The weight of a tree is proportional to the total of its edge weights, and the relevance of a tree is inversely related to its weight.

The example in Figure 1 illustrates that some links point towards the root of the tree, instead of away from the root as required by our model. For instance, the *Writes* relation has foreign keys to the *Paper* and *Author* relations, whereas we require paths from *Paper* to *Author*, traversing a foreign key edge in the opposite direction. However, we cannot simply regard the edges as undirected.

Ignoring directionality would cause problems because of "hubs" which are connected to a large numbers of nodes. For example, in a university database a department with a large number of faculty and students would act as a hub. As a result, many nodes would be within a short distance of many other nodes, reducing the effectiveness of proximity-based scoring.

To solve the problem, we create for each link/edge (u, v) a *backward edge* (v, u) with a different edge weight; we model the weight of (v, u) as directly proportional to number of links to v from the nodes of the same type as u. (Equations for computing the weights are presented in Section 2.2.) In the example from Figure 1, the backward edges ensure that there is a directed tree rooted at the paper, with a path to each leaf. To illustrate the effect of backward edge weights, let us return to the university department example. A forward edge from a student to her department and a back edge from the department to another student would form a path between each pair of students in the department. If there are more students in a department, the back edges would be assigned a higher weight, resulting in lower proximity (due to the department) for each pair of students, than if there are fewer students registered. In contrast, in the bibliographic database, papers (typically) have smaller numbers of authors, and the backward edge weights from *Paper* to *Writes* nodes would be less resulting in higher proximity between co-authors.

We may restrict the information node to be from a selected set of nodes of the graph; for example, we may exclude the nodes corresponding to the tuples from a specified set of relations, such as *Writes*, which we believe are not meaningful root nodes (this is similar to the scheme in [13]). In the example from Figure 1(B), let the keyword nodes be SunitaS, SoumenC and ByronD. These nodes, which are author nodes, have a relationship induced due to paper node ChakrabartiSD98. The tree shown in Figure 1(B) (with backward edges from the *Paper* node to the *Writes* nodes) would be a connection tree for the keyword nodes, with the paper node as the information node.

We incorporate another interesting feature, namely node weights, inspired by prestige rankings such as PageRank in Google [4]. With this feature, nodes that have multiple pointers to them get a higher prestige. In our current implementation we set the node prestige to the indegree of the node. Higher node weight corresponds to higher prestige. E.g., in a bibliography database containing citation information, if the user gives a query *Query Optimization* our technique would give higher prestige to the papers with more citations. As another example, in a TPCD database storing information about parts, suppliers, customers and orders, the orders information contains references to parts, suppliers and customers. As a result, if a query matches two parts (or suppliers, or customers) the one with more orders would get a higher prestige.

Node weights and tree weights need to be combined to get an overall relevance score as discussed in Section 2.3.

2.2. Formal Database Model

Based on the discussion thus far, our model comprises *nodes* with *node weight* and *edges* with *forward* and *backward* edge weights, and a *similarity measure* between relations.

Nodes/vertices: For each tuple $\mathcal{T}$ in the database, the graph has a corresponding node $u_{\mathcal{T}}$. We will speak interchangeably of a tuple and the corresponding node in the graph.

Node weights: Each node u in the graph is assigned a weight $N(u)$ which depends upon the prestige of the node. In our current implementation we set the node prestige to a function of the indegree of the node. Extensions to handle transfer of prestige (as is done, e.g., in Google's PageRank [4]) can be easily added to the model.

Edges: For each pair of tuples $\mathcal{T}_1$ and $\mathcal{T}_2$ such that there is a foreign key from $\mathcal{T}_1$ to $\mathcal{T}_2$, the graph contains an edge from $u_{\mathcal{T}_1}$ to $u_{\mathcal{T}_2}$ and a back edge from $u_{\mathcal{T}_2}$ to $u_{\mathcal{T}_1}$ (this can be extended to handle other types of connections).

Similarity between relations: Let $s(R_1, R_2)$ be the (generally asymmetric) similarity from relation R_1 to relation R_2 where R_1 is the referencing relation and R_2 is the referenced relation. The similarity $s(R_1, R_2)$ depends upon the type of the link from relation R_1 to relation R_2. $s(R_1, R_2)$ is set to infinity if relation R_1 doesn't refer to relation R_2.

Edge weights: In our model, the weight of a forward link along a foreign key relationship reflects the strength of the proximity relationship between two tuples and is set to 1 by default. It can be set to any desired value to reflect the importance of the link (small values correspond to greater proximity).

Consider two nodes u and v in the database. Let $R(u)$ and $R(v)$ be the respective relations that they belong to. The weight of the directed edge (u, v) depends on two conditions: whether the database has a link from u to v, and whether it has a link from v to u. Neither, one or both links may exist.

If (u, v) exists but (v, u) does not, we can simply assign the weight $s(R(u), R(v))$ to (u, v). If (u, v) does not exist and (v, u) does, according to our earlier arguments, we ought to assign weight $IN_v(u)\, s(R(v), R(u))$ to (u, v), where $IN_v(u)$ is the indegree of u contributed by the tuples belonging to relation $R(v)$.

If both (u, v) and (v, u) exist in the graph, we assign the weight $w(u, v)$ as the *minimum* of the two values, i.e.,

$$\min\{s(R(u), R(v)), IN_v(u)\, s(R(v), R(u))\}. \tag{1}$$

Other choices are possible. For instance, if one were to view the two weights as resistances in an electrical network, one may use the equivalent parallel resistance.

2.3. Query and Answer Model

Generally, a query consists of $n \geq 1$ search terms $t_1, t_2, \ldots, t_n$. The first step is to **locate** nodes matching search terms. A node is relevant to a search term if it contains the search term as part of an attribute value or metadata (such as column, table or view names). E.g., all tuples belonging to a relation named `AUTHOR` would be regarded as relevant to the keyword 'author'. For each search term t_i in the query we find the set of nodes S_i that are relevant to t_i. Let $S = (S_1, S_2, S_3, \ldots, S_n)$.

Extensions of the model to incorporate queries such as "author:Levy" which would require the keyword "Levy" to be in an author name attribute, can be easily incorporated. Approximate matching of keywords to words present in tuples can also be supported, by extending the model to incorporate node relevances. These features are not currently implemented in our prototype, and we omit further details.

An **answer to a query** is a rooted directed tree containing at least one node from each S_i. Note that the tree may also contain nodes not in any S_i and is therefore a Steiner tree. The relevance score of an answer tree is computed from the relevance scores of its nodes and its edge weights. (The condition that one node from each S_i must be present can be relaxed to allow answers containing only some of the given keywords.)

Figure 2 shows a sample result of a query containing the keywords *soumen* and *sunita* executed on the bibliographic database. Each result is a tree containing node tuples (including intermediate nodes) along with the resp. table names and column names. Indentation is used to depict the tree structure, and nodes containing keywords are distinguished from intermediate nodes by the color of the nodes.

Each answer tree has to be assigned a **relevance score**, and answers have to be presented in decreasing order of that score. Scoring involves a **combination** of relevance clues from nodes and edges. Node weights and edge weights provide two separate measures of relevance. We desire a final relevance score in the range [0,1]. We also wish to control the variation in individual weights so that a few nodes or edges with very large weights do not skew the results excessively. We therefore take the following approach.

- We scale individual node weights to $N_{\max}$, the maximum node weight in the graph. We can additionally depress the scale using logarithms (as with 'IDF' weighting in Information Retrieval); accordingly, the normalized score $N\mathit{score}(v)$ of a node v is defined as $N(v)/N_{\max}$ or $\log(1 + N(v)/N_{\max})$ respectively. These are both scale-free quantities in $[0, 1]$ (if log is to base 2).

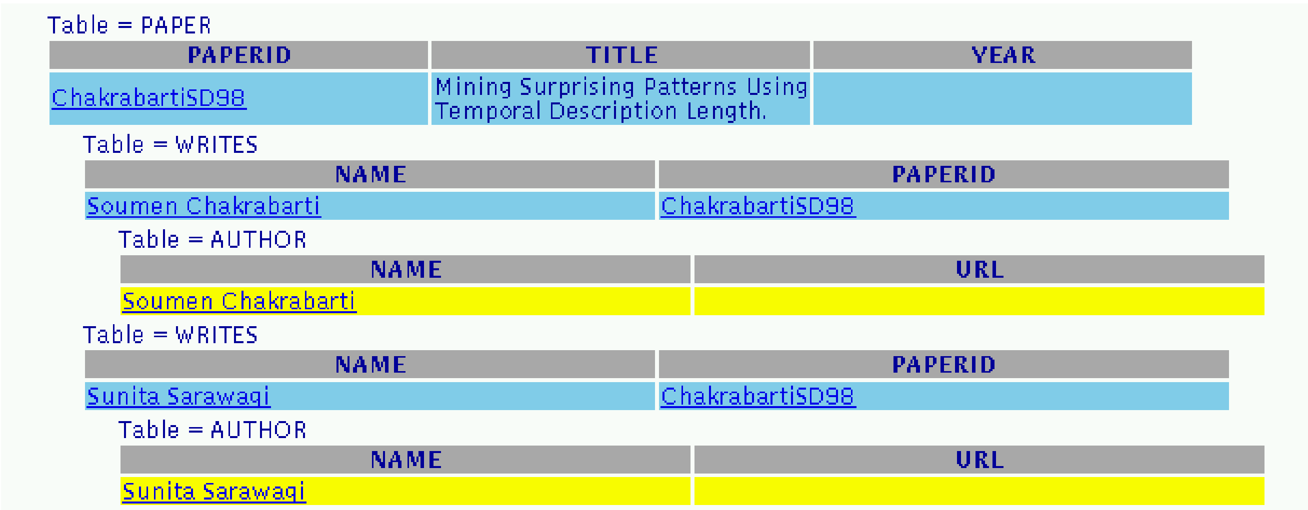

Figure 2. Result of query "soumen sunita"

To get the overall node score $Nscore$, we take the average of the node scores. To favor meaningful root nodes, and to reduce the effect of intermediate nodes, we consider only leaf nodes (containing the keywords) and the root node when computing the average. A node containing multiple search terms is counted as many times as the number of search terms it contains, to avoid giving extra weight to trees with separate nodes for each keyword.

- We get the normalized edge score $Escore(e)$ of an edge by dividing the edge weight by $w_{\min}$, the minimum edge weight in the graph, to make it scale-free, and may additionally depress the scale by defining the edge score of e as $\log(1 + w(e)/w_{min})$.

  The overall edge score is then defined to be $Escore = 1/(1 + \sum_e Escore(e))$, since we wish to give lower relevance to large trees. This quantity is also in the range $[0, 1]$.

- Finally, we can combine the overall edge score and node score, to get an overall relevance score, either by addition or by multiplication; in both cases, a factor λ controlling their relative weightage. The additive combination uses the formula $(1 - \lambda)Escore + \lambda Nscore$, while the multiplicative combination uses the formula $Escore * Nscore^{\lambda}$.

There are a total of eight combinations, since we have three options (for edge score, node score and combination) each of which can take two values. In our evaluation we discarded three combinations: those that involve log scaling and multiplication as these scores tended to become quite small, and compared the remaining combinations.

While inspired by standard IR weighting and smoothing practice, the choices and parameters above are somewhat ad-hoc, but this appears to be inescapable in all related systems that we have reviewed [7, 17].

3. Searching for the Best Answers

The computation of minimum Steiner trees is already a hard (NP complete) problem, and is made complicated by node weight considerations, required to compute the overall relevance of a tree. We are interested in not just the most relevant tree, but also in other trees with high relevance scores, since they may be part of what the user is searching for. We also wish to generate answers incrementally to avoid generating answers of low relevance that the user may never look at.

In this section, we present an outline of the *backward expanding search algorithm* which offers a heuristic solution for incrementally computing query results. Complete details can be found in the full version of the paper [3].

We assume that the graph fits in memory. This is not unreasonable, even for moderately large databases, because the in-memory node representation need not store any attribute of the corresponding tuple other than the RID. The only other in-memory structure is an index to map RIDs to the graph nodes. Indices to map keywords to RIDs can be disk resident. As a result the graphs of even large databases with millions of nodes and edges can fit in modest amounts of memory.

Given a set of keywords, first we find, for each keyword term t_i, the set of nodes, S_i, that are relevant to the keyword by using disk resident indices on keywords.

Let $\mathcal{S} = \cup S_i$. The backward expanding search algorithm concurrently runs $|\mathcal{S}|$ copies of Dijkstra's single source shortest path algorithm, one for each keyword node n in $\mathcal{S}$, with n as the source. The copies of the algorithm

```
Global: #Keywords = n; Keywords: {t_1, t_2, ..., t_n},
Keyword node sets: {S_1, S_2, ..., S_n}, S = ∪S_i
IteratorHeap = φ; OutputHeap = φ
For each keyword node, n ∈ S
   Create a single source shorest path iterator with n
   as the source and put it in IteratorHeap
   ordered on the distance of the first node it will output
while IteratorHeap is not empty and more results required
   Iterator = remove top iterator from IteratorHeap
   v = Get next node from Iterator
   If Iterator has more nodes to output
      Insert Iterator again in IteratorHeap ordered on
      the distance of the next node it will output
   if v is not visited before by any iterator then
      for i = 1 to n: Create v.L_i and set v.L_i = φ
   CrossProduct = origin × ∏_{i≠j} v.L_j
   where origin is the origin of Iterator and origin ∈ S_i
   /* CrossProduct is empty if any v.L_j is empty */
   Insert origin in v.L_i
   for each tuple ∈ CrossProduct
      create ResultTree from tuple
      /* ResultTree is rooted at v and contains a path
         from v to each origin node in tuple */
      if root of ResultTree has only one child
         continue /* duplicate result */
      if OutputHeap is full
         Output and remove top result from OutputHeap
      Insert ResultTree into OutputHeap
         ordered by its relevance score
```

Figure 3. Backward Expanding Search

are run concurrently by creating an iterator interface to the shortest path algorithm, and creating an instance of the iterator for each keyword node.

Each copy of the single source shortest path algorithm traverses the graph edges in reverse direction. The idea is to find a common vertex from which a forward path exists to at least one node in each set S_i. Such paths will define a rooted directed tree with the common vertex as the root and the corresponding keyword nodes as the leaves. The tree thus formed is a connection tree and root of the tree is the information node.

In the example from Figure 1(B), let the keyword nodes be SunitaS, SoumenC and ByronD. The algorithm will have three shortest path iterators one starting from each keyword node. All the iterators will visit paper node ChakrabartiSD98. Thus, the algorithm generates a connection tree rooted at the paper node (the information node in the tree) with the keyword nodes as the leaf nodes. The tree shown in Figure 1(B) is the connection tree with all edges directed away from the paper node. (Note that each edge in the figure has a corresponding opposite edge in the graph but is not shown.) Note, that the algorithm may generate more results as it may detect other information nodes, e.g., if the authors have coauthored more than one paper.

Figure 3 shows (high-level) pseudocode for the backward expanding search algorithm. In each iteration, the algorithm picks an iterator whose next vertex to be output is at the least distance from the source vertex of the iterator. (The distance measure can be extended to include node weights of nodes matching keywords.)

To find information nodes and the corresponding connection trees incrementally, within each vertex (visited by any iterator), say v, we maintain a nodelist $v.L_i$ for each search term t_i. $v.L_i \subset S_i$ and is empty initially. Consider an iterator started from a keyword node, say $u \in S_i$, visiting node v. Some other iterators might have already visited node v and the keyword nodes corresponding to those iterators are already in resp. $v.L_i$'s. All possible connection trees rooted at node v and containing keyword nodes from $v.L_i$'s are already generated. Thus we need to generate the new connection trees containing node u. We generate a cross product of node u with the rest of the nodelists $\{u \times \prod_{i \neq j} v.L_i\}$ and each cross product tuple corresponds to a connection tree rooted at node v. Trees whose root has only one child are discarded, since the tree formed by removing the root node would also have been generated, and would be a better answer.[1] After generating all connection trees, we insert node u in list $v.L_i$.

The connection trees generated by the algorithm are only approximately sorted in the increasing order of their weights. (The weight of a tree is the sum of the weights of the edges.) The relevance of a tree is computed using the tree weight and the node weights as discussed in Section 2.3. Currently, node weights are not considered while generating the connection trees. As a result, trees may not be generated in exact decreasing relevance order.

We could generate all connection trees and then sort them in decreasing relevance order, but this would increase computation costs and leads to a greatly increased time to generate initial results. To avoid these overheads, as a heuristic, we maintain a small fixed-size heap of generated connection trees. The heap is ordered on the relevance of the trees. We keep adding trees to the heap as they are generated, without outputing them. When the heap is full, and we want to add a new tree, we output the tree of highest relevance and replace it in the heap. When all answers have been generated, the remaining trees in the heap are output in decreasing order of relevance. While this heuristic does not guarantee that the trees are generated in decreasing order, we have found it works well even with a reasonably small heap size.

The algorithm may generate trees that are isomorphic

[1] Generally the smaller tree would have higher relevance, although if the root of the larger tree has a higher node weight it is possible for the larger tree to have higher relevance.

modulo direction; that is, their undrected versions are same. We call such trees as duplicate trees. They represent the same result, except with different information nodes. We retain only the one with the highest relevance and discard the rest. We mainatin a list of all the results generated so far to allow duplicate detection. When a new result is generated, if a duplicate is in the heap, and its relevance is smaller than the that of the new result, we remove the duplicate from the heap and insert the new result into the heap. This can happen since results are not necessarily generated in decreasing order of relevance. In fact, a duplicate of the result might have already been output; in that case we discard the new result even if its relevance is higher that a duplicate that was output earlier.

4. Browsing

The BANKS system provides a rich interface to browse data stored in a relational database. The browsing system automatically generates browsable views of database relations and query results; no content programming or user intervention is required.

Every displayed foreign key attribute value becomes a hyperlink to the referenced tuple. In addition, primary key columns can be browsed backwards, to find referencing tuples, organized by referencing relations (a specific referencing relation can be selected by the user).

Each table displayed comes with a variety of tools for interacting with data.

- Columns can be projected away (dropped)
- Selections can be imposed on any column
- For foreign key columns, clicking on "join" results in the referenced table being joined in, and its columns also displayed. This eliminates the need for explicitly writing join queries for the normal case of foreign key join. The join feature can also be used in the other direction, from a primary key to a referencing foreign key.
- Results can be grouped-by on a column; this results in only the distinct values for that column being displayed. The user can click on any of the values to see the tuples associated with that value.
- Tuples in the displayed table can be sorted by a specified column

Controls for these operations can be accessed by clicking on the column names in the table header. In addition, displayed data is paginated, and schema browsing is supported.

Figure 4 shows the result of browsing the thesis database starting with the *student* relation, using a pop-up menu on the roll number attribute to effect a join with the *thesis* relation and dropping columns. The join is made possible

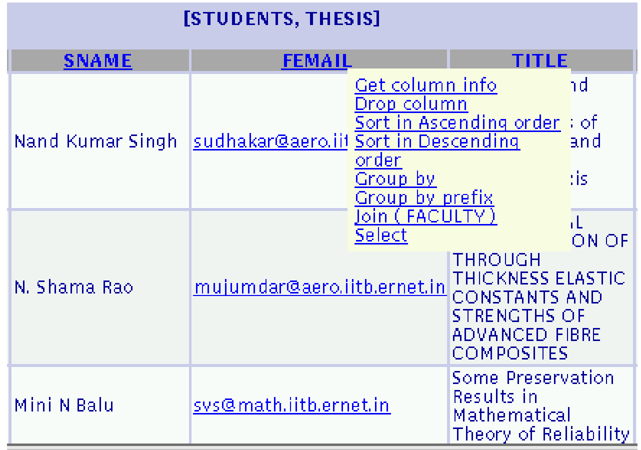

Figure 4. Sample browsing session.

since the *thesis* relation has a foreign-key attribute referencing the *student* relation. A sample pop-up menu is shown for the *femail* attribute which references the *faculty* table. Underlined attribute values are hyperlinks.

BANKS **templates** provide several predefined ways of displaying any data. Template instances are customized, stored in the database, and given a hyperlink name, which is used to access the template. The BANKS system currently provides four types of templates:

- Cross-tabs (similar to OLAP cross-tabs).
- The group by template provides for hierarchical view of data, by specifying a sequence of grouping attributes. For example, grouping a student relation by department and program attributes initially displays all departments; clicking on a department shows all programs in the department, and clicking on a program then shows all students in that program in the selected department.
- Folder views are similar to grouping, but are modeled after the folder view of files and directories supported in many environments such as Windows Explorer.
- The graphical interface template permits information to be displayed in *bar chart*, *line chart* or *pie chart* format. Hyperlinks are provided on the graphical data via HTML image maps; clicking on a bar of a bar chart, or a slice of a pie chart shows tuples with the associated value.

Another interesting feature of templates is that they can be composed together in a hyperlinked, visual manner. The action associated with a hyperlink may be scripted to take the user to another template, instead of showing the detailed tuples.

5. Experience and Performance

We have implemented BANKS using servlets, with JDBC connections to an IBM Universal Database. We have experimented with two datasets. The first dataset contained a part of the DBLP information, represented in structured relational format. There are about 100000 nodes and 300000 edges in the resultant BANKS graph. The other dataset had information about Masters and Phd dissertations in IIT Bombay, and its graph had thousands of nodes and tens of thousands of edges.

There are no agreed-upon benchmarks for evaluating ranking algorithms in this domain. To work around this, we selected data sets that we as academics and database researchers can relate to, and picked queries that illustrated different ways of querying this information (e.g. keywords from two authors who are coauthors, authors who have a common coauthor, an author and a title, keywords from titles alone, and so on). *Across all the queries (with proper parameter settings, discussed later) we found the system returning the most intuitive answers ahead of less intuitive ones in almost all cases.*

5.1. Anecdotes

We give a few examples of queries and the answers returned by BANKS. For the query "Mohan" on the DBLP database, C. Mohan came out at the top of the ranking, with Mohan Ahuja and Mohan Kamat following. This was due to the prestige conferred by the *writes* relation which had many tuples for these authors. The query "transaction" returned Jim Gray's classic paper and the book by Gray and Reuter as the top two answers.

As another example, on the thesis database, the query "computer engineering" returned the Computer Science and Engineering department with a higher relevance than a number of thesis that had these two words in their title, since the larger number of references to the department gave it a higher node weight. The query "sudarshan aditya" returned a thesis written by Aditya whose advisor is Sudarshan.

On the DBLP database, the query "soumen sunita" returned the answer shown in Figure 2. The query "seltzer sunita" returned Stonebraker as the root, with connections to Sunita and Seltzer through papers co-authored by Stonebraker with each of them separately. Without log scaling on edges, this answer got a lower rank than other less meaningful answers with large trees, since the backward edge from Stonebraker to the *Writes* tuples has a very high weight due to the large number of papers written by Stonebraker.

5.2. Space and Time

For a bibliographic database with 100K nodes and 300K edges, memory utilization was around 120 MB. Java implementations are notorious for wasting space. The graph currently takes about 2 minutes to load initially, with almost all the time spent in Java structure creation. We expect much smaller memory utilization and loading/running time with a properly tuned Java/C implementation.

Once the database graph is loaded, queries take about a second to a few seconds for most queries on the bibliographic database. Overall, even with a prototype which has not been tuned carefully, it is feasible to use BANKS for moderately large databases.

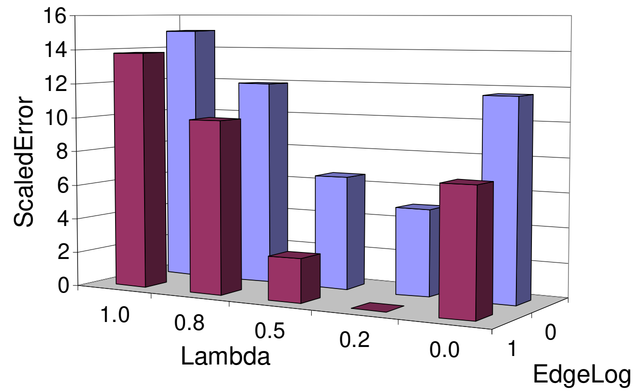

Figure 5. Error scores vs. parameter choices.

5.3 Effect of Parameters

Our performance evaluation was conducted using 7 different queries whose form was outlined earlier. For each query we chose answers that we felt were the most meaningful, and we call these the ideal answers; there were an average of 4 such answers per query. We ran our algorithm on each query, with different combinations of the parameters (edge and node scoring functions and score combination), stopping at 10 answers. For each query, for each parameter setting, we computed the absolute value of the rank difference of the ideal answers with their rank in the answers for that parameter setting. The sum of these rank differences gives the raw error score for that parameter setting. We scaled the scores to set the worst possible error score to 100. We considered answers to be the same if their trees were the same, even if the roots were different. For answers that were missing at a parameter setting, the rank difference was assumed to be 11 (one more than the number of answers examined).

Figure 5 shows error scores against λ and log-scaling of edge weights (EdgeLog=1 represents log-scaling). The following conclusions can be drawn from our performance study.

- It was important to keep the effect of node ranking relatively small, but non-zero. Setting λ to 0.2 with log scaling of edge weights did best, with an error score of 0.0, while setting λ to 0.5 with log scaling of edge weights did almost as well with error scores of around

3. Setting λ to 1 (ignore edge weights) did the worst, with error scores of around 15, followed by $\lambda = 0.8$ and $\lambda = 0$ (ignore node weights) with scores of between 8 and 12, with and without log scaling of edge weights respectively.

Note that the absolute values of the error scores are relatively small, even when we ignore edge weights. This is because results are generated in increasing order of edge weight and then sorted by relevance score using a limited buffer, and our heuristic of discarding trees where the root node has only one child eliminates larger trees even if their root nodes have high node weight.

- Reducing the edge weight range by log-scaling was important, otherwise back edges from some popular nodes had a high weight and resulted in some intuitively correct answers getting a very poor relevance rank. With good settings for other parameters, using log scores reduced the error score by around 5.
- The "mode" of score combination (additive/multiplicative) has almost no impact on the ranking (and as a result on error scores), although the absolute values of the relevance scores were different.
- For node weights, log scaling gave the same ranking as no log scaling on our examples, although we can construct scenarios where log scaling does better.

In conclusion, the rankings are relatively stable across different choices of parameter values, but $\lambda = 0.2$ coupled with log scaling of edge weights does best.

6. Related Work

BANKS is closely related to the DataSpot system [6, 12, 13]. (Dataspot is now part of the Mercado software system.) In particular, the model of query answers as rooted trees corresponds to the DataSpot model, where the roots are called "fact nodes". DataSpot also computes relevance scores for trees, and returns trees of maximum relevance. However, the details of the underlying graph formalisms differ. BANKS currently assumes a model where only those references corresponding to equivalence edges in DataSpot are explicitly represented. Since edges in our model can have attributes such as type and weight, we can model containment (as in DataSpot and in nested XML) simply as edges of a new type. (We are currently working on adding XML support to BANKS.) The BANKS technique of assigning weights to back edges based on indegree as well as its use of node weights have no counterpart in DataSpot. The use of node weights based on prestige has proved critical in Web search, and our anecdotal evidence shows their importance in the context of database search as well. BANKS also takes the effect of metadata queries into account, which is not made explicit in DataSpot.

The idea of proximity search in databases represented as graphs was also proposed by Goldman et al. [7]. They support queries of form *find object near object*. They restrict results to tuples from one relation near a set of keywords, whereas we permit results to be structured as trees which helps explain how we arrive at an answer. Unlike BANKS, they do not consider node and edge weighting techniques.

EasyAsk (`www.easyask.com`) is another commercial system that provides natural language search (including keyword search) on data stored in relational databases. EasyAsk does a variety of tasks such as approximate word matching and natural language understanding. However, details of how they handle keyword queries are not publicly available.

Web search provides another natural application where the best response may comprise a graph of connected pages rather than a single page. Like us, Li et al. [9] couch this problem in terms of Steiner trees. However, in their formulation, the graphs are not directed, and they do not handle queries that exploit meta-data. Proximity is the primary concern in their setting, whereas BANKS combines proximity with link-based prestige. Unlike BANKS, they do not consider structured data sources such as databases.

Another system for keyword search and browsing of databases is *Mragyati*, by Sarda and Jain [14]. Their implementation does not handle paths of length greater than two. Their ranking system can use user-specified criteria, but the default ranking system uses indegree, which is one of many criteria in BANKS.

Miller et al. [10] describe a system for querying and browsing of data stored in databases. They concentrate on dynamically generating multiple hierarchical views for users to drill down to find required data. They allow selection predicates but do not consider keyword queries. Object oriented database browsers such as OdeView [1] and Pesto [5] provide navigation by clicking on object references, but do not support keyword search. Our system also provides more powerful browsing facilities. BBQ [11] presents an interface for blended browsing and querying but querying in BBQ requires the user to know the database schema. Shneiderman et al. [15] [16] have developed systems to display chemical elements, search for homes or movies, and so on, based on the concept of dynamic queries. Their systems focus on graphical user interface and do not consider keyword queries, unlike BANKS.

Hulgeri et al [8] provide a more detailed survey of related work in this area; our graph model and query model are presented in that paper, but many details of the model, and details of query evaluation algorithms and browsing are new to this paper.

7. Ongoing and Future Work

We are currently extending the BANKS system to handle browsing and keyword searching of XML data. We plan to implement `attribute:keyword` queries such as `author:Levy`. We are investigating authority transfer (a form of spreading activation), wherein nodes pointed to by heavy nodes (perhaps via user feedback) become heavier. We are considering implementing some form of approximate matching, such as `concurrency approx(1988)` to look for papers about concurrency published around 1988.

We also want to summarize the output, i.e., group the output tuples into sets that have the same tree structure, and allow the user to look for further answers with a particular tree structure.

We are exploring support for external links, such as HTML HREFs, to aid in browsing. Such support is particularly useful when integrating information from multiple databases. Other planned system features include authorization mechanisms to selectively expose data to different users.

Query evaluation with keywords matching metadata can be relatively slow, since a large number of tuples may be defined to be relevant to the keyword. This problem also arises with non-metadata keywords that match large number of nodes. We are working on techniques to speed up such queries by not performing backward search from large numbers of nodes, and instead searching forwards from probable information nodes corresponding to more selective keywords.

8 Conclusions

We have developed BANKS, an integrated browsing and keyword querying system for relational databases. BANKS allows users with no knowledge of database systems or schema to query and browse relational database with ease. BANKS greatly reduces the effort involved in publishing relational data on the Web and making it searchable. Examples of the types of data that could be published using BANKS include organizational data, bibliographic data and product catalogs.

We have proposed a framework for answering keyword queries, and implemented an algorithm to find query answers incrementally. We have evaluated our prototype in terms of speed and meaningfulness of answers using academic and bibliographic databases. Our observations are that BANKS is practical to use on moderately large databases, and that the results are intuitive and useful.

Acknowledgments: We wish to thank B. Aditya, Urmila Kelkar, Megha Meshram and Parag for implementing some parts of the BANKS system, and helping with the performance evaluation.

References

[1] R. Agrawal, N. H. Gehani, and J. Srinivasan. OdeView: The graphical interface to Ode. In *Proc. of ACM SIGMOD*, pages 34–43, 1990.

[2] P. Bailey, N. Craswell, and D. Hawking. Dark matter on the Web. In *Poster Proceedings, 9th World-Wide Web Conference*, 2000.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. Technical report, Indian Institute of Technology, Bombay, November 2001.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7), 1998.

[5] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. PESTO : An integrated query/browser for object databases. In *Proc. of the Int'l Conf. on VLDB*, pages 203–214, 1996.

[6] S. Dar, G. Entin, S. Geva, and E. Palmon. DTL's DataSpot: Database exploration using plain language. In *Proc. of the Int'l Conf. on VLDB*, pages 645–649, 1998.

[7] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proc. of the Int'l Conf. on VLDB*, pages 26–37, 1998.

[8] A. Hulgeri, G. Bhalotia, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword search in databases. *IEEE Data Engineering Bulletin*, 24(3):22–31, Sept. 2001.

[9] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal. Retrieving and organizing Web pages by "information unit". In *10th World-wide Web Conference*, 10, pages 230–244, 2001.

[10] R. J. Miller, O. G. Tsatalos, and J. H. Williams. DataWeb: Customizable database publishing for the web. *IEEE MultiMedia*, 4(4):14–21, 1997.

[11] K. D. Munroe and Y. Papakonstantinou. BBQ: A visual interface for integrated browsing and querying of XML. In *Visual Database Systems*, May 2000.

[12] E. Palmon. Associative search method for heterogeneous databases with an integration mechanism configured to combine schema-free data models such as a hyperbase. United States Patent Number 5,740,421, Granted April 14, 1998, filed in 1995. Available at `www.uspto.gov`, 1998.

[13] E. Palmon and S. Geva. Associative search method with navigation for heterogeneous databases including an integration mechanism configured to combine schema-free data models such as a hyperbase. United States Patent Number 5,819,264, granted October 6, 1998, filed in 1995. Available at `www.uspto.gov`, 1998.

[14] N. L. Sarda and A. Jain. Mragyati: A system for keyword-based searching in databases. Report No. cs.DB/011052 on CORR (http://xxx.lanl.gov/archive/cs), 2001.

[15] B. Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, 11(6):70–77, 1994.

[16] B. Shneiderman. Dynamic queries, starfield displays, and the path to Spotfire. Feb. 1999. http://www.cs.umd.edu/hcil/spotfire/.

[17] R. Weiss, B. Vélez, M. A. Sheldon, C. Nemprempre, P. Szilagyi, A. Duda, and D. K. Gifford. HyPursuit: A hierarchical network search engine that exploits content-link hypertext clustering. In *Proc. of ACM Hypertext*, pages 180–193, 1996.