DBXplorer: A System for Keyword-Based Search over Relational Databases

Sanjay Agrawal
Microsoft Research
sagrawal@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Gautam Das
Microsoft Research
gautamd@microsoft.com

Abstract

Internet search engines have popularized the keyword-based search paradigm. While traditional database management systems offer powerful query languages, they do not allow keyword-based search. In this paper, we discuss DBXplorer, a system that enables keyword-based search in relational databases. DBXplorer has been implemented using a commercial relational database and web server and allows users to interact via a browser front-end. We outline the challenges and discuss the implementation of our system including results of extensive experimental evaluation.

1. Introduction

Internet search engines have popularized keyword-based search. Users submit keywords to the search engine and a ranked list of documents is returned to the user. An alternative to keyword search is *structured search* where users direct their search by browsing classification hierarchies. Both models are tremendously valuable – success of both keyword search and the classification hierarchy are evident today.

A significant amount of the world's enterprise data resides in relational databases. It is important that users be able to seamlessly search and browse information stored in these databases as well. Searching databases on the internet and intranet today is primarily enabled by *customized* web applications closely tied to the schema of the underlying databases, allowing users to direct searches in a structured manner. Examples of such searches within, say a bookseller's database may be "Books → Travel → Lonely Planet → Asia", or "Books → Travel → Rough Guides → Europe".

While such structured searches over databases are no doubt useful, unlike the documents world, there is little support for *keyword search over databases*. Yet, such a search model can be extremely powerful. For example, we may like to search the Microsoft intranet on 'Jim Gray' to obtain *matched rows*, i.e., rows in the database where 'Jim Gray' occur. Note that such matched rows may be found in more than one table, perhaps even from different databases (e.g., address book and mailing lists). Our goal is to enable such searches without necessarily requiring the users to know the schema of the respective databases. Yet, today's customized web applications as described above and traditional SQL applications *require* knowledge of the schema.

Enabling keyword search in databases that does not require knowledge of the schema is a challenging task. Note that one cannot apply techniques from the documents world to databases in a straightforward manner. For example, due to database normalization, logical units of information may be fragmented and scattered across several physical tables. Given a set of keywords, a matching row may need to be obtained by joining several tables *on the fly*. Secondly, the physical database design (e.g., the availability of indexes on various database columns) needs to be leveraged for building compact data structures critical for efficient keyword search over relational databases. In this paper we describe *DBXplorer*, an efficient and scalable *keyword search utility for relational databases*. The task of building DBXplorer gives rise to several research questions that we address in this paper.

Alternatives in Symbol Table Design: Traditional information retrieval techniques for enabling keyword search in document collections use data structures such as *inverted lists* [2] that efficiently identify documents containing a query keyword. A straightforward mapping of this idea to databases is a *symbol table* that stores information at row level granularity, i.e., for each keyword we keep the list of rows that contains the keyword. Alternative symbol table designs are possible where we can leverage the physical design of the database. For example, if a column has an index then we only need column level granularity, i.e., for each keyword, we only store the list of columns where they occur. The above approach can result in vastly reduced space requirement and improved search performance. In this paper, we study the trade-offs among these various alternatives.

Symbol Table Compaction: We introduce a novel technique that leverages commonality of keywords among database columns to compress symbol tables. This

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 **IEEE**

technique is used in conjunction with hashing and other known compression techniques.

Efficient Search across Multiple Tables: Often, the results of a query are matching rows that span multiple tables. For example, consider the schema graph of a book retailer's database *Pubs* in Figure 1, where the nodes are the tables and the edges represent foreign key relationships. Suppose a user is searching for books on 'programming' by 'Ritchie'. The two keywords are not present in a single row in any one table. The required information is distributed across the tables *Authors*, *Books* and *AuthorsBooks*. The rows need to be generated by joining tables on the fly by *exploiting the schema as well as content of the database*.

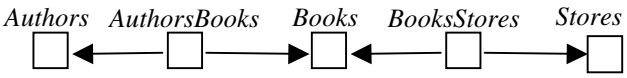

Figure 1. Pubs database

Efficient Generalized Matches using SQL: When an attribute value is a string containing multiple keywords, retrieving rows where a keyword matches a substring (e.g., LIKE "*%kwd%*") cannot exploit an index lookup on the attribute. In such cases, full text search functionality is necessary for efficiency. We show a novel alternative for doing such matches using SQL. We explore the applicability and limitations of our scheme.

DBXplorer supports *conjunctive* keyword queries, i.e., retrieval of only documents that contain *all* query keywords. This is, in fact, the most widely used paradigm for Internet search.

We have implemented DBXplorer using commercially available Microsoft SQL Server 2000 database server and Microsoft IIS web server. It communicates with databases using the standard ODBC interface, and thus can be supported over virtually any relational database. Our design ensured that DBXplorer leverages the functionality of the relational engine effectively. DBXplorer is currently deployed on our corporate intranet, and several databases have been enabled for keyword search using this tool.

The rest of this paper is organized as follows. In Section 3, we present an overview of DBXplorer. Sections 4 and 5 describe the preprocessing component responsible for creating the symbol table. Section 6 describes the search component that answers keyword searches once the symbol table has been built. Section 7 discusses extensions needed for generalized keyword matches described above. Section 8 presents experiments that demonstrate the effectiveness of our solution. An appendix containing screen shots of user interactions with DBXplorer is included to provide a better feel for the front-end.

2. Related Work

Keyword-based search is a well studied problem in the world of text documents and Internet search engines. *Inverted lists* are common data structures used for solving keyword queries [2, 5, 15, 23, 24]. An interesting post-search activity is the ranking of results [2, 14]. Our work differs from canonical use of inverted lists because we need to generate hits within a database that span *multiple tables*, as materializing all table joins and publishing each as a document (and using a text search engine) is not a scalable solution. This has ramifications for symbol table design as will be discussed in Section 4.

The approach in [5] addresses the problem of keyword search over XML documents. It parses XML documents to generate and load inverted file information (i.e., a map of values to individual rows) into a relational database. Our design provides an alternative where symbol tables map keywords to columns that have available indexes. The work in [6] addresses the problem of *proximity* search over semi-structured stores. In contrast, our core focus is on finding *exact* matches in a *multi-relation* database that contains *all* keywords specified in the query, requiring us to study design alternatives for symbol tables as well as to develop techniques for join tree enumeration.

The *Telegraph* FFF engine searches for facts and figures from selected sites on the Internet, and allows them to be combined and analyzed in complex ways [18]. Since our work allows websites to expose their tabular information for enabling keyword search, the FFF search mechanism at the websites that provides facts and figures may be augmented by DBXplorer technology.

The search component of DBXplorer bears resemblance to work on *universal relations* [20], where a database is viewed as a single universal relation for querying purposes, thus hiding the complexity of schema normalization. The challenge in the universal relation approach is to map a selection query over the universal relation to a SQL query over the normalized schema. Although certain aspects of our search algorithm (such as join trees, see Section 6.1) are similar to universal relations concepts (such as window functions, see [20]), an important difference is that keyword searches have to deal with the additional complexity that the *names of columns* in the selection conditions are not known.

DataSpot [3] is a commercial system that supports keyword-based searches by extracting the content of the database into a *hyperbase*. Thus, this approach duplicates the content of the database, which makes data integrity and maintenance difficult. Microsoft's *English Query* [11] provides a natural language interface to a SQL database. However, unlike the keyword-based approach, it "guesses" a single SQL statement that best fits a query expressed in a natural language.

Most major commercial database vendors allow a full text search engine (e.g., [10, 12]) to be invoked while processing SQL (that is extended by specialized predicates). However, such engines cannot by themselves identify matching rows that result from joining multiple stored tables on-the-fly (see Section 6).

3. Overview of DBXplorer

Given a set of query keywords, DBXplorer 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. Although for lack of space, we discuss only the case where there is a single database, our techniques extend to keyword search over *multiple databases*.

3.1. Overview of Publish and Search Steps

Publish: A database (or a desired part of it) is enabled for keyword search through the following steps.
Step 1: A database is identified, along with the set of tables and columns within the database to be published.
Step 2: Auxiliary tables are created for supporting keyword searches. The most important structure is a *symbol table* **S** that is used at search time to efficiently determine the locations of query keywords in the database (i.e., the tables, columns, rows they occur in).

Search: Given a query consisting of a set of keywords, it is answered as follows.
Step 1: The symbol table is looked up to identify the tables, and columns/rows of the database that contain the query keywords.
Step 2: 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 they are connected in the schema, i.e., there is a sub-tree (called a *join tree*) in the schema graph that contains these tables as nodes (and possibly some intermediate nodes).
Step 3: For each enumerated join tree, a SQL statement is constructed (and executed) that joins the tables in the tree and *selects those rows that contain all keywords*. The final rows are ranked and presented to the user.

3.2. System Architecture

DBXplorer has been deployed on real databases from the intranet within Microsoft. For its implementation, we leverage IIS web server and Active Server Pages (ASP) [8]. The main Publish and Search components are packaged as two separate COM (Component Object Model [9]) objects. The publish component provides interfaces to (a) select a database, (b) select tables/columns within the database to publish, and (c) modify/remove/maintain the publication. For a given set of keywords, the search component provides interfaces to (1) retrieve matching databases from a set of published databases, and (2) selectively identify tables, columns/rows that need to be searched within each database identified in step (1). The specific interfaces for the latter include (i) for a given set of keywords, find all the matching tables/columns, (ii) for a given set of keywords, find all rows in the database that contain all of the keywords.

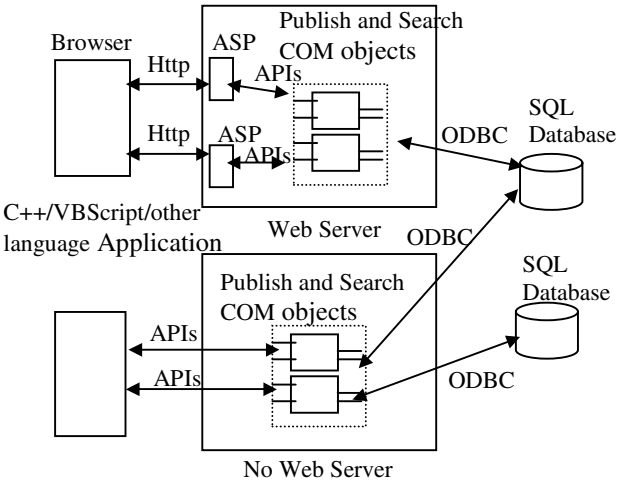

Figure 2. Architecture of DBXplorer

Packaging these components as COM objects enables them to be used in a variety of applications. For example, Figure 2 shows two different applications that are connecting to the Publish and/or Search COM objects – the first uses web server/Active Server Pages (ASP) and second is an application in C++/VB or any scripting language. For the former, two separate ASP pages are used as front-ends for publishing and search which in turn call the interfaces provided by the respective COM components. This model allows use of a standard web browser to publish any database at a web server. Similarly, for search, the user connects to the search ASP using a browser and issues a keyword-based query to get matching rows. The system also allows one to search multiple databases simultaneously. (See the appendix for screenshots of the system.)

4. Design Alternatives for Symbol Table

In this section we present and analyze different symbol table designs. We only consider the *exact match* problem; i.e., where each keyword in the query must

match the value of an attribute in a row of a table. We defer handling of more generalized matches to Section 7. The symbol table is the key data structure used to look up the respective locations of query keywords in the database. An important design consideration is deciding the *location granularity* i.e., for a given keyword, what information needs to be stored in the symbol table to identify the location of the keyword in the database. The two interesting granularity levels are: (a) *column level granularity* (*Pub-Col*), where for every keyword the symbol table maintains the list of all database columns (i.e., list of table.column) that contain it, and (b) *cell level granularity* (*Pub-Cell*), where for every keyword the symbol table maintains the list of database *cells* (i.e., list of table.column.rowid) that contain it.

Some choices of the granularity levels are not quite as interesting. For example, our experiments have shown that *row level granularity* symbol tables (that maintain list of rows that contain a keyword) have little advantage over cell level granularity as far as the size of the symbol table is concerned, yet certain functionalities (e.g., to "un"-publish a column, i.e., to stop making the column available for keyword search) are harder to implement because column information is absent.

There are several factors that influence the appropriate granularity level to adopt: (a) space and time requirements for building the symbol table, (b) effect on keyword search performance, and (c) ease of symbol table maintenance. We discuss these factors next.

4.1. Space and Time Requirements

The symbol table size is a critical factor in system performance; larger symbol tables increase I/O costs during search. Pub-Col symbol tables are usually much smaller than Pub-Cell symbol tables. This is because unlike the latter, if a keyword occurs multiple times in a column (corresponding to different rows), no extra information needs to be recorded in Pub-Col. Our experiments on test databases show orders of magnitude differences between the two symbol table sizes (see Section 8.1). The time to build Pub-Col symbol table is also correspondingly less, since unlike Pub-Cell we only need to record the *distinct* values in a column.

4.2. Keyword Search Performance

As will be discussed in Section 6, each keyword search query results in a set of SQL statements, which are then executed to retrieve matching rows. Search performance depends on the efficient generation and subsequent execution of these SQL statements. SQL generation requires that the tables and columns where the keywords may occur be known. This is achieved by retrieving symbol table entries. We now discuss the effect of alternative symbol table designs on SQL generation.

Consider the *o_orderpriority* column in the *Orders* table in a 100MB TPC-H database. In this database, the *Orders* table has 150,000 rows and *o_orderpriority* has 5 distinct values. While using a Pub-Cell table, a search on a value in *o_orderpriority* can lead to approximately 30,000 cells (i.e., 150,000/5, assuming uniform data distribution) being retrieved from a Pub-Cell symbol table. To retrieve the matching rows, SQL queries will need to be generated that explicitly refer to the rowids corresponding to the 30,000 cells identified above from the symbol table. In contrast, with a Pub-Col table we will retrieve only one entry *o_orderpriority* (the column name) and the corresponding SQL has the simple form *select * from Orders where Orders.o_orderpriority =$keyword*. Of course, Pub-Col is effective only if database indexes are available for the published columns (e.g., on *o_orderpriority*) so that the generated SQL statements can be efficiently executed.

4.3. Ease of Maintenance

Maintenance of symbol tables as data in databases change is an important consideration. For insertions, Pub-Col is easier to maintain as it requires an update only if the insertions cause new values to be introduced in some column data. In contrast, Pub-Cell needs to be updated for every inserted row. Likewise, every deleted row does not necessarily cause an update in a Pub-Col table. Updates are handled in a similar fashion. One can use triggers or time stamps to update the symbol table with changes in underlying data.

4.4. Summary

The Pub-Col symbol table alternative is almost always better than the Pub-Cell table, unless certain columns do not have indexes. In general, a *hybrid* symbol table is needed where the granularity is tied to the physical database design: if an index is available for a column, we publish the column contents with Pub-Col granularity, otherwise we publish it with Pub-Cell granularity.

5. Storing Symbol Tables in Databases

5.1. Pub-Col Representation

A naive implementation of the Pub-Col symbol table is to store it as a relation with two attributes: Keyword and ColId (assume that each database column has been mapped to a unique ColId). However, a simple but effective compression strategy is to keep *hashed values* of keywords in the symbol table (i.e., attribute Keyword is replaced by attribute HashVal), since storing keywords

directly is wasteful as they may be potentially long strings of varying lengths. Despite this optimization, there are significant opportunities for further compression, as we show in this section.

We show how the Pub-Col symbol table may be compressed by *taking advantage of the occurrences of keywords among multiple columns*. We present two algorithms in this section. The compression algorithm, *FK-Comp*, works as follows. If the set of values in column c_1 is a subset of the values in another column c_2 due to a key-foreign key relationship, we retain only a single hash table entry for keywords common to both as (*keyval,* c_1), since the foreign key constraint can be used to infer presence of *keyval* in c_2.

HashVal	ColId
v_1	c_1
v_2	x
v_3	x
v_4	x
v_5	c_2

Table 1. Uncompressed hash table

HashVal	ColId
v_1	c_1
v_2	c_1
v_3	c_1
v_4	c_1
v_2	c_2
v_3	c_2
v_4	c_2
v_5	c_2

Table 2. Compressed hash table

NewColId	ColId
x	c_1
x	c_2

Table 3. ColumnsMap table

We now describe a more general compression algorithm, *CP-Comp* that can take advantage of situations where pairs or sets of columns share common keywords, but are not necessarily tied by foreign key relationships, e.g., {OldEmployer, CurrentEmployer}, {PurchaseDate, ShipDate, ReceiveDate}. The algorithm is based on *bipartite clique partition* techniques [4]. Let HashVal = $\{v_1, v_2, \ldots, v_n\}$ be the set of distinct hash values of all keywords, and ColId = $\{c_1, c_2, \ldots c_m\}$ be the set of all distinct columns. We map the symbol table **S** to a bipartite graph **H** having two node sets, HashVal and ColId, where every row (v, c) in table **S** corresponds to an edge in **H**. The two key steps in *CP-Comp* are: (a) Partition **H** into a *minimum* number of bipartite cliques (a bipartite clique is any subgraph of **H** with a maximal number of edges). (b) Compress each clique. We begin by describing step (b) first. Consider the uncompressed hash table in Table 1. The corresponding bipartite graph contains the clique with the two node sets $\{v_2, v_3, v_4\}$ and $\{c_1, c_2\}$, and all edges between them. We compress this clique by deleting all its edges, introducing a new "super" node x that represents the set of columns $\{c_1, c_2\}$, and adding edges from each node in $\{v_2, v_3, v_4\}$ to x. The resultant hash table is shown in right of Table 2. Note that we also keep a separate table ColumnsMap to keep track of the relationship between the derived "super" node and the original nodes $\{c_1, c_2\}$ (Table 3).

We now describe step (a). Partitioning a bipartite graph into a minimum number of cliques is NP-complete [4]. However, CP-Comp uses an efficient heuristic that works as follows. For each hash value v, we compute the set of columns that contain one or more keywords that map to the hash value v. Let $\{ColId_1,\ldots,ColId_p\}$ be the set of *distinct* column subsets computed in this manner. We partition HashVal into subsets $\{HV_1,\ldots,HV_p\}$, where keywords mapping to hash values in HV_i can only occur among columns in $ColId_i$. Note that each $\{HV_i, ColId_i\}$ pair induces a clique in the graph **H**. We can compress these cliques as described in step (a).

Algorithm Computing a Pub-Col Symbol Table
Inputs: A database
Outputs: A symbol table **S** and ColumnsMap table
//Compute hash table **S**:
  Set **S** to empty
  Scan database, and for each keyword K in column c
    Insert (hash(K), c) into **S** if it does not already occur
//Compress **S** using Algorithm CP-Comp:
  Set table ColumnsMap to empty
  Compute $\{ColId_1,\ldots, ColId_p\}$ and $\{HV_1,\ldots, HV_p\}$
    For i = 1 to p
      If $|HV_i|*|ColId_i| > |HV_i| + |ColId_i|$
        Remove from **S** all entries involving HV_i
        Create artificial column x_i
        For each v in HV_i, insert (v, x_i) into **S**
        For each c in $ColId_i$, insert (c, x_i) into ColumnsMap
  Output **S** and ColumnsMap

Figure 3. Constructing Pub-Col symbol table

The main steps of constructing a Pub-Col symbol table construction are summarized in Figure 3. Maintenance of the compressed symbol table is similar to the uncompressed Pub-Col table but extra updates are necessary for the ColumnsMap table.

5.2. Pub-Cell Representation

A Pub-Cell symbol table can be stored as a two column table consisting of (Hashval, Cellid) pairs. However, such a normalized representation turns out to be especially disadvantageous for Pub-Cell since the same keyword may occur in many cells. Retrieving all these locations for SQL generation during search can become very slow as potentially many rows of Pub-Cell symbol table needs to be accessed. Thus, we adopt a more efficient design in which the symbol table has two attributes, HashVal and CellIdList. The latter is a variable length column in which we keep a *list* of the CellIds of all the cells in which a keyword appears (a non first normal

form representation). As a result, retrieving all locations for a keyword is achieved by looking up a single row from the Pub-Cell symbol table.

We cannot compress a Pub-Cell symbol table using the CP-Comp algorithm described above, since a cell contains only one keyword. However, algorithms that compress relational implementations of inverted lists (e.g., see [15]) do apply to Pub-Cell. Incidentally, CP-Comp can be pipelined with algorithms in [15] for additional compression of Pub-Col.

6. Finding Matches for Keyword Search

In this section we discuss the search component and focus only on the *exact match* case. Let $\{K_1, K_2, \ldots, K_k\}$ be the keywords specified in a query. Recall from Section 3.1 that keyword search has three steps. In the first step, the symbol table is searched (using generated SQL) to identify the database tables, columns/cells that contain at least one of the keywords in the query. The next two steps are that of enumerating join trees and identifying matching rows that are described in detail below.

6.1. Enumerating Join Trees

This step is similar for all symbol table granularities. Let MatchedTables be the set of database tables that contain at least one of the query keywords. If we view the schema graph **G** as an undirected graph, this step enumerates *join trees*[1], i.e., sub-trees of **G** such that: (a) the leaves belong to MatchedTables and (b) together, the leaves contain all keywords of the query. Thus, if we join the tables that occur in a join tree, the resulting relation will contain all potential rows having all keywords specified in the query. This important step filters out a large number of spurious join scenarios from being passed on to the subsequent step of the search.

For example, consider the schema graph **G** in Figure 4 over five tables. Let the query keywords be $\{K_1, K_2, K_3\}$. The black nodes represent the MatchedTables set, while the rest are white. Assume that K_1, K_2 and K_3 all occur in T_2 (in different columns), K_2 occurs in T_4, and K_3 occurs in T_5. The four possible join trees are shown on the right (including the singleton node, T_2). In contrast, for example the tree induced by $\{T_4, T_3, T_5\}$ cannot be a join tree, as these tables do not contain all keywords.

We outline our algorithm for enumerating join trees. For simplicity of exposition, we assume **G** itself is a tree. We first prune **G** by repeatedly removing white leaves, until all leaves are black (this resembles *ear removal* operations for computing window functions in universal relations [20]; see also [7, 21]). The resulting tree, **G'**, is guaranteed to contain all potentially matching join trees. Our next task is to enumerate all *qualifying sub-trees* of **G'**, i.e., sub-trees such that all keywords in the query occur among the black nodes of the sub-tree. For efficient enumeration, we adopt a heuristic for picking the *first* node of the candidate qualifying sub-trees as follows: We pick the keyword that occurs in the *fewest* black nodes of **G'**. We now do breadth-first enumeration of all sub-trees of G' starting from each of the black nodes identified above and check if it is a qualifying sub-tree. Using this heuristic considerably reduces the number of trees enumerated. Note that if we cannot assume that **G** is a tree (i.e., if it contains cycles), the join tree enumeration involves bi-connected component decomposition [17] of **G**, followed by the enumeration of join trees on a possibly cyclic schema graph [13, 16]. We omit further details due to lack of space.

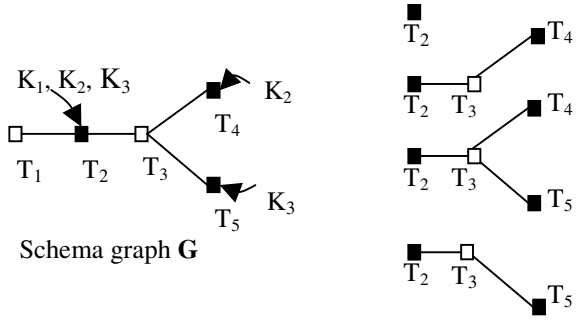

Figure 4. Join trees

6.2. Searching for Rows

The input to this final search step is the enumerated join trees. Each join tree is then mapped to a single SQL statement that joins the tables as specified in the tree, and selects those rows that contain all keywords. In fact, this is the only stage of the search where the database tables are accessed. For a Pub-Col symbol table, the generated SQL statement will have selection conditions on columns, whereas for a Pub-Cell symbol table, the selection conditions will involve rowids (and for a hybrid table, the selection condition will involve a mix of both types of conditions). The execution efficiency depends on several factors, e.g., availability of column indexes for the Pub-Col based approach. We observe that there may be commonalities among the generated SQL statements for a given keyword search query, with potential applications of multi-query optimization for further efficiency.

The retrieved rows are ranked before being output. Our approach is to rank the rows by the number of joins involved (ties broken arbitrarily); the reasoning being that joins involving many tables are harder to comprehend.

[1] Traditionally, the term *join tree* refers to the ordering of join operations determined by the query optimizer for a given query. We have overloaded the term to refer to a kind of subgraph (as defined above) of the schema where edges depict key foreign key relationship.

This has parallels with certain ranking methods used in document retrieval (e.g., documents in which keywords occur close to one another are ranked higher than documents in which keywords are far apart). Since our enumeration algorithm generates join trees in order of increasing size (due to breadth first enumeration), the join tree enumeration step can be pipelined and thus followed immediately by the SQL generation corresponding to the join tree. We summarize the main steps of search in Figure 5.

Algorithm SEARCH
Inputs: A query consisting of keywords $K_1, K_2, \ldots, K_k$
Outputs: All database rows matching all keywords, including rows derived by joining tables on the fly
//Search symbol table:
  Look up symbol table **S** to determine the tables, columns or cells containing query keywords
//Enumerate join trees:
  Compute **G'** from **G** by ear removal operations
  Enumerate join trees in **G'**
//Search for rows:
  For each join tree (in increasing size), construct and execute SQL statement to retrieve matching rows

Figure 5. Search

7. Supporting Generalized Matches

In this section we discuss more general kinds of keyword matches. Specifically, we focus on the important case of *token matches* where the keyword in the query matches only a token or sub-string of an attribute value (for text string attributes, e.g., addresses, where we may wish to retrieve rows by specifying only a street name).

7.1. Token Matches

As a simple example, consider a database with a table T as shown in Table 4. Let the hash values of the searchable tokens i.e., 'string', 'ball' and 'round' be 1, 2 and 3 respectively (we ignore stop words such as 'this", 'is' etc.). During publishing (for all symbol table granularities) we tokenize each cell, hash and store each distinct token along with appropriate location information in the respective symbol table.

Consider searching with a Pub-Col symbol table. If a query keyword is 'string', this symbol table tells us that it occurs in column T.C. For a join tree that has T.C as a node, the generated SQL will need to have clauses with substring predicates such as WHERE T.C LIKE '%string%'. Since traditional B+ tree indexes cannot be used for index seeks to exploit such predicates. As an alternative, most recent commercial database systems support *full-text indexes* that enable token search in text columns (e.g., Microsoft SQL Server [12]). If a full-text index is present for column T.C, the generated SQL will have clauses such as WHERE CONTAINS(C, 'string'), which can be efficiently executed. In this section, we present a novel technique that uses some pre-computation but can perform token searches using B+ indexes that are supported on all traditional SQL databases. The trade-offs and the range of this applicability are also studied experimentally in Section 8.

Table 4. Database table T

RowId	C
1	this is a string
2	this string
3	this is a ball
4	no string
5	any ball is round

Table 5. Pub-Prefix table

Hash Val	Col Id	Prefix
1	c	th
1	c	no
2	c	th
2	c	an
3	c	an

The search component for Pub-Cell symbol tables remains the same as in the exact match case (See Section 6). Essentially, these symbol tables mimic some of the functionalities of full-text indexes. However, recall from Section 4.1 that Pub-Cell symbol tables may be large and could rival the size of the database itself.

We now present *Pub-Prefix*, a method that efficiently enables token match capabilities by exploiting available traditional B+ tree indexes. It is based on the following crucial observation: B+ tree indexes can be used to retrieve rows whose cell matches a given prefix string. That is, clauses of the form WHERE T.C LIKE 'P%K%' where P is any prefix string can be efficiently computed. During publishing of a database, for every keyword K, we keep in the symbol table the entry (hash(K), T.C, P) if there exists a string in column T.C which (a) contains a token K, and (b) has prefix P. For example if we publish the database table shown in Table 4, the resulting Pub-Prefix symbol table is shown in Table 5 (assuming we stored two character long prefixes).

Consider searching for the keyword 'ball'. Looking up this keyword in Table 4 returns the prefixes 'th' and 'an', and the subsequent SQL will contain clauses such as WHERE (T.C LIKE 'th%ball%') OR (T.C LIKE 'an%ball%'). Such clauses can be efficiently evaluated with traditional B+ tree indexes (in the above example, rows 3 and 5 will be retrieved from the database). Pub-Prefix tables can be compressed using the CP-Comp algorithm, except that instead of hash values we use (hash value, prefix) pairs.

We expect the search performance of Pub-Prefix method to be comparable to Pub-Cell method when the column width is small (e.g., columns such as *name* and *address* which are typically less than 100 characters). For columns with strings of several hundred characters (e.g.,

product reviews) Pub-Cell can outperform Pub-Prefix significantly. The Pub-Prefix table size depends largely on the column data and the prefix length to store in symbol table. An interesting issue is determining an appropriate prefix length. As the prefix length is increased, its discriminating abilities (and symbol table size) increases, and in the limit the prefix method degenerates to the Pub-Cell method. On the other hand, as the prefix length is decreased, its discriminating abilities (and the symbol table size) decreases, and in the limit the prefix method degenerates to the Pub-Col method. We evaluate different prefix lengths through experiments in Section 8.4. Note that we can tune Pub-Prefix even further by allowing *different* prefix lengths for different tokens. We are presently investigating these extensions in our design.

In summary, if a full-text index is available, use Pub-Col with the full-text index. Instead, if only a traditional index is available and the column width is small, use Pub-Prefix, otherwise use Pub-Cell.

7.2. Other Generalized Matches

We are currently investigating the feasibility of implementing other generalized match capabilities within our system. Several of them appear to only require straightforward adaptations of corresponding techniques from the information retrieval domain. Allowing matches with variants of query keywords (e.g., 'run' and 'running') can be addressed by standard information retrieval techniques such as *stemming* [2]. The Pub-Cell based technique is unaffected by stemming, except that stemming is applied prior to storing keywords in the symbol tables and to search keywords as well. The Pub-Col table is more complicated since to find all variants of a keyword, they need to be *explicitly* mentioned in the WHERE clause of the generated SQL for looking up matching rows, e.g., Book.title = "cat" or Book.title = "cats". For most words in English, an explicit disjunction can be avoided by using LIKE, e.g., Book.title LIKE "cat%". But, a general solution is more complex. For Pub-Prefix, each returned row will need to be stemmed to check if it contains acceptable variations of the search keywords. We are currently investigating the problems of adding a broader set of matching capabilities, such as synonyms, fuzzy matches, and disjunctive (and more general Boolean) keyword queries.

8. Experiments

We present the results of an experimental evaluation of the publishing and search techniques presented in this paper (Pub-Col, Pub-Cell and Pub-Prefix). In particular we show that:

- Pub-Col table is compact compared to Pub-Cell. Search performances for the two techniques are comparable when the number of rows selected by keywords is small. Pub-Col has superior performance when keywords are not very selective.
- Pub-Col scales linearly with data size, and is independent of data distribution, both in publishing time and symbol table size. Search performance scales with data size and number of search keywords.
- Pub-Prefix table is compact compared to Pub-Cell. Search performance of Pub-Prefix is significantly better than Pub-Col when full-text indexes are not present. For small width columns (order of tens of characters), search performance of Pub-Prefix is comparable to Pub-Cell and Pub-Col with full-text indexes.

Setup: The experiments are on a 450 MHz 256 MB Intel P-III machine. We used four databases, three of which are from the intranet of Microsoft Corporation.
Synthetic database: TPC-H data of sizes 100 to 500 MB (TPCHx denotes TPC-H data of size x MB).
Real databases: USR, a 130 MB employee address book database with 19 tables; ML, a 375 MB mailing lists database with 38 tables that contains information such as owners and participants; and KB, a 365 MB database with 84 tables that contains information on articles and help manuals on various shipped products.

8.1. Symbol Table Granularity

We compare the publishing and search performance of two techniques: Pub-Col and Pub-Cell.

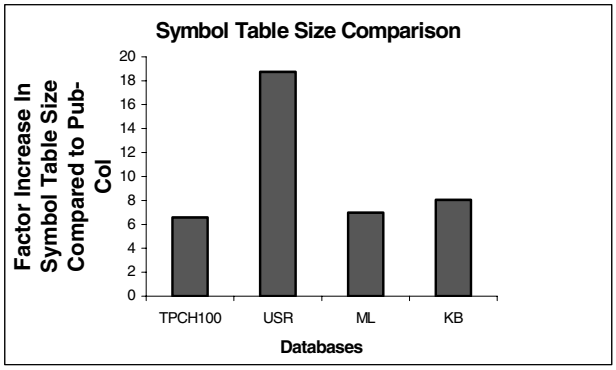

Figure 6. Symbol table size

Symbol Table Size: Figure 6 demonstrates that the Pub-Col symbol table is an order of magnitude smaller than Pub-Cell. For the USR database, Pub-Cell is larger by a factor of 19. Recall that USR has information about employees and fields such as last name and job titles that are shared by a number of people. Even for the synthetic TPCH100 database where strings are arbitrarily

generated, Pub-Cell is 7 times larger. This shows that Pub-Col produces the most compact symbol tables. We also observed that the size of the Pub-Col symbol table is a small fraction of the base data size (e.g., for USR it is 3%, for ML and KB it is 10%, and for TPCH100 it is 25%).

Publishing Time: Populating the symbol table takes around 50% of the total publishing time; the rest of the time includes scanning the database and processing the data to store in the symbol table. Publishing time for Pub-Cell is about 7 times that for Pub-Col (Figure 7). This is primarily because much less data needs to be fetched from the database by DBXplorer to generate the Pub-Col symbol table (see Section 4.2).

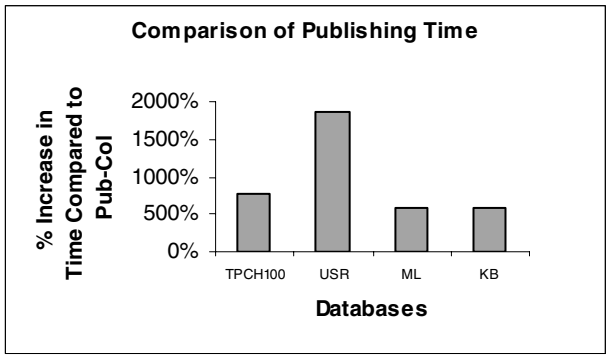

Figure 7. Symbol table building time

Search Performance: Two workloads consisting of 100 queries are generated for each database. The number of keywords in a given query is randomly generated between 1 and 5. The keywords themselves are randomly selected from the symbol table of the underlying database. We denote the workloads for a database D as D-WKLD-FEW (consists of keywords that select fewer than 10 records) and D-WKLD-LARGE (consists of keywords that select more than 100 records).

Figure 8 shows the average end-to-end query time (normalized with respect to Pub-Col) for the different techniques. We observe that Pub-Col and Pub-Cell have similar performances for TPCH100-WKLD-FEW and USR-WKLD-FEW. SQL generation time is almost the same for the two techniques, as very few symbol table entries (needed for SQL generation) match the keywords. SQL execution time is also almost same due to the presence of relevant database indexes.

However for TPCH100-WKLD-LARGE and USR-WKLD-LARGE, Pub-Col has a superior performance compared to Pub-Cell (see Figure 8). For WKLD-LARGE the symbol table look-up time for Pub-Cell increases while remaining relatively unchanged for Pub-Col. Pub-Cell takes about twice the look-up time compared to Pub-Col for both USR-WKLD-LARGE and TPCH100-WKLD-LARGE. This is because Pub-Cell causes a much larger amount of data (all the matching row-ids) to be retrieved from the symbol table for subsequent SQL generation. The additional time is due to larger symbol table and slower character data processing (compared to integer data processing in Pub-Col).

We had warmed up SQL Server's buffers with the symbol table in the above experiments. However if we start from cold buffers, the look-up time increases by another 20% for Pub-Cell, as the larger symbol table size contributes to more I/O. The increase is much smaller (about 5%) in the look-up time for Pub-Col. If we have multiple users accessing different databases concurrently, having a smaller symbol table can make a significant difference in search performance.

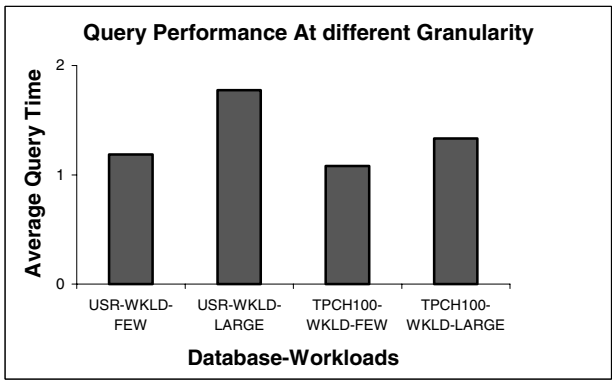

Figure 8. Query performance

This establishes that it is a better strategy (in both publishing space and search time) to use Pub-Col, especially when some keywords might match a large number of rows in the databases. It is important to note that if column indexes are not available, search performance of Pub-Col can degrade rapidly. In that case, one should use Pub-Cell for the columns.

8.2. Scalability of Pub-Col

Data Size and Distribution: We use TPCH100-WKLD-FEW (described in Section 8.1) as workload. We use the TPC-H uniform database and vary the data size from 100 MB to 500 MB. In these databases the number of distinct keywords is proportional to the data size.

The space required by the symbol table for TPC-H data varies almost linearly with data size. The publishing time for TPC-H data also increased almost linearly with data size. The publishing time is dominated by the time required to scan the data and populate the symbol table; both these operations vary linearly with data size.

Figure 9 shows that the average query execution time (normalized with respect to time taken for TPCH100) increases very slowly as the data size is increased to 500 MB. This is due to a small increase in symbol table look-up time (recall that the symbol table sizes increase

proportionately with data size) and relatively more expensive SQL execution to fetch the matching rows from the database.

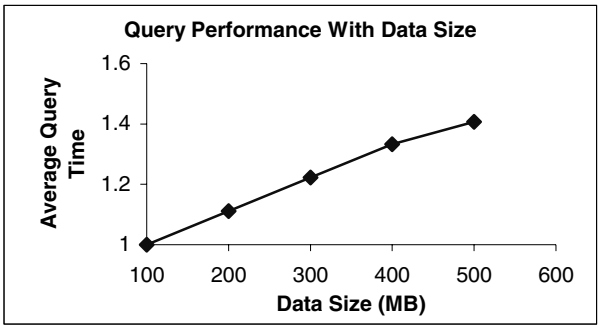

Figure 9. Search performance for TPCH

To study the impact of data distribution on publish and search, we generated 100 MB TPC-H data with Zipfian distribution [22] (zipf = 1,2,3) and compared with TPC-H uniform data. The distinct keywords in all these databases were the same. The publishing overhead is almost the same (same symbol table sizes and similar creation times). The search performance is also very comparable (within 3%). This is expected, as publish and search are not sensitive to the distribution of values as long as the distinct data values are the same.

Number of Keywords in Search: We show that search scales with the number of query keywords. We also show that when databases have complex schema, the search continues to perform well. We generated five workloads of 100 queries each for each database. We varied the number of keywords in each workload from 2 to 10 in steps of 2. The keywords in each workload were selected randomly from the underlying database.

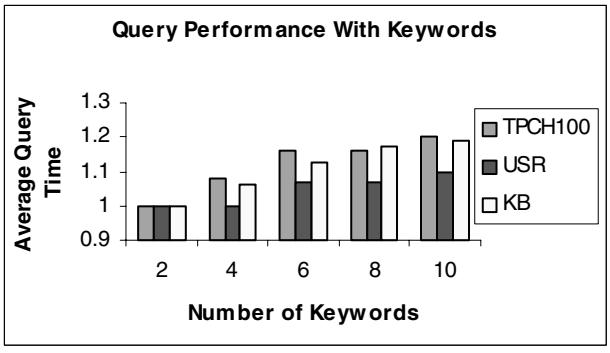

Figure 10. Search on TPCH100, USR, and KB

Figure 10 shows that the search performance (normalized with respect to time taken for two keywords) for TPCH100, USR and KB as the number of keywords are increased to 10. For all three databases, search time increases slowly. This is expected as (a) symbol table look-up time remains almost unchanged, and (b) number of join trees explored (and SQL statements generated to get matching rows) is small even for KB that has a relatively complex schema. The results on ML were similar to KB, thus we do not report them here.

8.3. Effectiveness of Compression Techniques

We compare the compression schemes FK-Comp and CP-Comp[2] (discussed in Section 5.1). Figure 11 shows that for ML and KB, the compressed symbol table size is around 50% of the table size generated by Pub-Col. The compression achieved for FK-Comp is consistently less compared to CP-Comp for all databases.

In ML (and KB), there are several columns that share common data that is not captured by FK-Comp (e.g., email address of a particular mailing list's owner is also embedded in list creation request description) that allow the data to be compressed significantly. We observed that the time to compress symbol table using FK-Comp and CP-Comp was similar. We also observed that compression added a negligible overhead on search performance. The reason is because additional look-up into a relatively small ColumnsMap table (typically memory resident) is very fast.

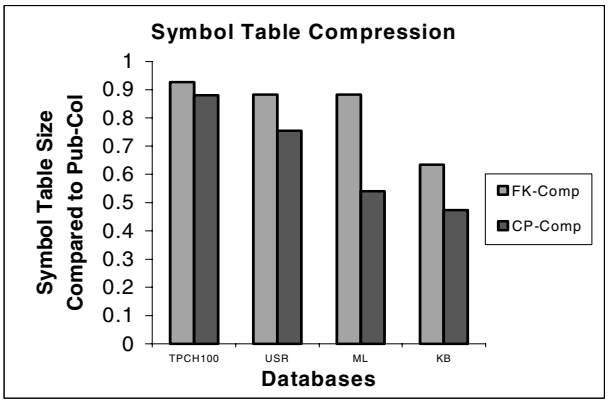

Figure 11. Quality of compression techniques

8.4. Effectiveness of Pub-Prefix Method

We evaluate the space requirements and search performance of Pub-Prefix (see Section 7.1). We compare it to Pub-Cell, Pub-Col with a full-text index present on the data (referred to as *Pub-Col-FTS*), and Pub-Col without any full-text indexes present.

Search Performance Comparisons: We generated a workload consisting of 100 randomly selected keywords

[2] CP-Comp incorporates a heuristic to eliminate from consideration columns that do not contribute significantly to compression. Our experiments indicated that only considering columns that together contribute 80% of the total estimated compression improves the running time of CP-Comp remarkably.

from a character column of width 64 bytes in the KB database. The total size of the data in that column was 12.5 MB. Figure 12 shows average search time (normalized with respect to Pub-Col) when prefix length is varied from 2 to 16. We observe that Pub-Prefix gives the best performance at prefix length 8. This is because as the prefix length is increased, the discriminating power of a prefix increases and so does the number of prefixes associated with a keyword. This induces additional disjunctions in the subsequently generated SQL query. After a certain limit, for such queries, the optimizer resorts to a scan of the underlying table instead of an index seek. Thus the average query execution time increases. We observe similar behavior for a character column of length 40 in USR, where the best prefix length is 6. It is important to note that the nature of the curve that we obtain is generic; the specific optimum point depends on the underlying column data.

The performance of Pub-Prefix is comparable to Pub-Cell and Pub-Col-FTS for both KB and USR (within 5%). Pub-Prefix outperforms Pub-Col (without full-text indexes) by an order of magnitude (factor of about 30 for KB and 8 for USR). However, for ML where the data length in each cell is large (300 bytes) and many cells have common prefixes, Pub-Col-FTS has the best performance (5% better than Pub-Cell), while the performance of Pub-Prefix degrades to linear scan.

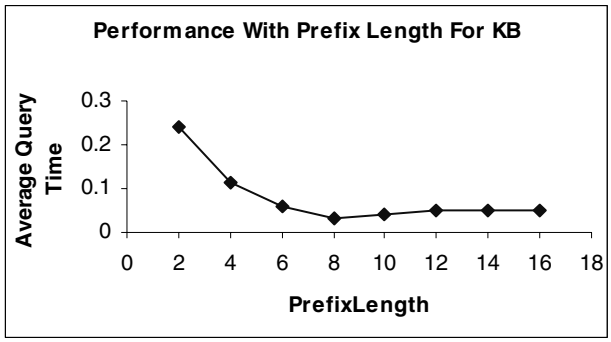

Figure 12. Search time versus prefix length

Space Comparisons: We analyze the space requirements of Pub-Prefix with varying prefix lengths. We observe that the Pub-Prefix table is an order of magnitude smaller than the Pub-Cell symbol table. As the prefix length is increased from 2 to 16 the symbol table size increased linearly from 1.6 MB to 3.4 MB. Pub-Cell symbol table size is comparable to the base data size (~12.5 MB). These experiments show that when the width of the column is small, Pub-Prefix leads to a more compact symbol table than Pub-Cell and has comparable search performance. Note that Pub-Prefix requires a B+ tree column index on the column.

Acknowledgments

We are very grateful to Vivek Narasayya and Venky Ganti for their insightful comments on the algorithms and presentation of this paper

References

[1] S. Abiteboul, Querying Semi-Structured Data, ICDT, 1997.
[2] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, ACM Press, 1999.
[3] S. Dar, G. Entin, S. Geva, E. Palmon, DTL's DataSpot.,VLDB, 1998.
[4] T. Feder, R. Motwani, Clique partitions, Graph Compression and Speeding-Up Algorithms, STOC, 1991.
[5] D. Florescu, I. Manolescu, Integrating Keyword Search into XML Query Processing, 9th WWW Conf., 2000.
[6] R. Goldman, N. Shivakumar, S. Venkatasubramanian, H. Garcia-Molina., Proximity Search in Databases, VLDB, 1998.
[7] M. H. Graham, On the Universal Relation. Technical Report, Univ. of Toronto, 1979.
[8]http://www.microsoft.com/ windows2000/server/evaluation/features/web.asp
[9] http://www.microsoft.com/com/
[10] http://www.microsoft.com/ntserver/web/exec/feature/IndexServerSummary.asp
[11] http://www.microsoft.com/sql/productinfo/eqmain.htm
[12] http://www.microsoft.com/sql/productinfo/fulltext.htm
[13] S. Kapoor, H. Ramesh, Algorithms for Enumerating all Spanning Trees of Directed and Undirected Graphs, SIAM J. Computing, 1995.
[14] J. M. Kleinberg, Authoritative Sources in a Hyperlinked Environment, SODA, 1998.
[15] S. Melnik, S. Raghavan, B. Yang, H. Garcia-Molina, Building a Full Text Index for the Web, http://dbpubs. stanford.edu/pub/2000-29, 2000.
[16] G. J. Minty, A Simple Algorithm for Listing all Trees of a Graph, IEEE Trans. on Circuit Theory, 1965.
[17] R. E. Tarjan, Depth-first Search and Linear Graph Algorithms, SIAM J. of Computing, 1972, 146-160.
[18] The Telegraph System, http://fff.cs.berkeley.edu/
[19] TPC Benchmark H (Decision Support) Revision 1.1.0., http://www.tpc.org/
[20] J. D. Ullman, Principles of Databases and Knowledge-Base Systems, Vol II, Computer Science Press, 1989.
[21] C. T. Yu, M. Z. Ozsoyoglu, An Algorithm for Tree-Query Membership of a Distributed Query, IEEE COMPSAC, 1979.
[22] G.E. Zipf, Human Behavior and the Principle of Least Effort, Addison-Wesley Press, Inc, 1949.
[23] N. Ziviani, E. Silva de Moura, G. Navarro, R. Baeza-Yates, Compression: A Key for Next Generation Text Retrieval Systems, Computer 33(11): 37-44, 2000.
[24] J. Zobel, A. Moffat, K. Ramamohanarao, Inverted Files versus Signature Files for Text Indexing, ACM TODS, 1998.

Appendix: User Interfaces for DBXplorer

We illustrate typical user interactions with DBXplorer via screen shots. Consider a query {'Livia', 'Karsen', 'Computer'}; perhaps the user is looking for a book by the author. The system first returns the set of databases that contain the given keywords (Figure 13), along with a brief description of each matching database (auto-generated from schema description). This aids the user in selecting a specific database to be explored next.

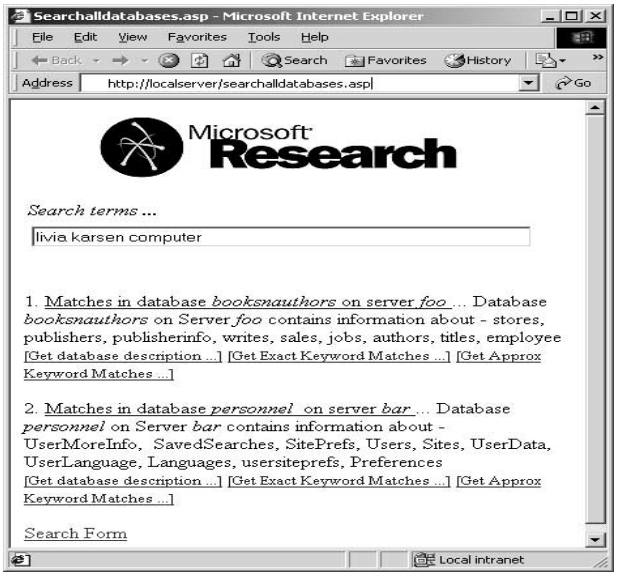

Figure 13. Matching databases

In the next step, the user explores matches within a selected database. For each keyword, the system returns a list of table and column pairs where the keyword occurs (top half of Figure 14). The system also enumerates a list of subsets of tables (join trees) such that the relation obtained by joining these tables may yield rows containing all keywords (bottom half of Figure 14).

The user has the option of selecting one or more (or even all) of the join trees for the system to explore. The system responds by presenting a ranked list of the final matching rows (Figure 15).

Figure 14. Join trees

The system also offers browsing capabilities whereby the user can explore further details of the retrieved rows by following links into related areas within the database. Figure 15 shows a book written by 'Livia Karsen' that has navigational links ("incoming" and "outgoing" links that represent foreign-key relationships) to stores and publishers of this particular book.

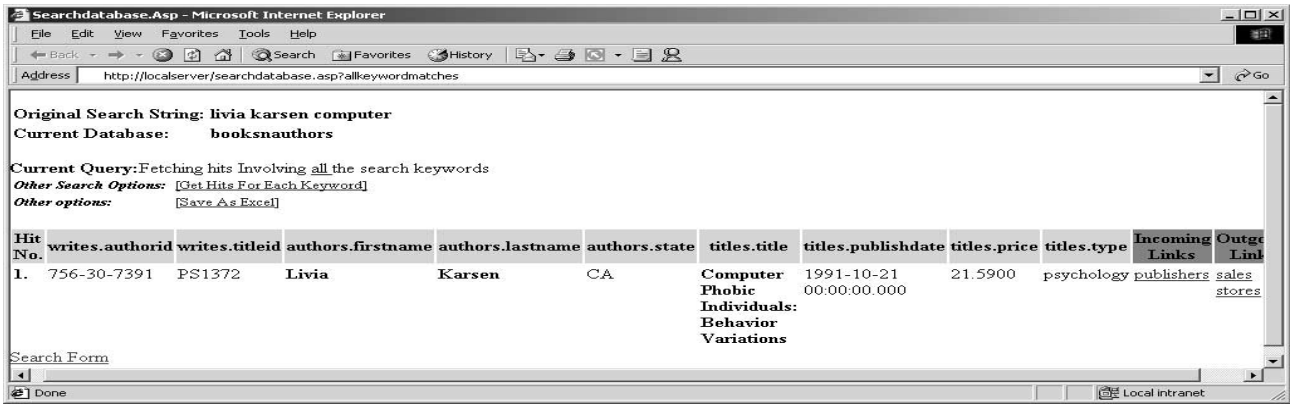

Figure 15. Matching rows