Indexing Relational Database Content Offline for Efficient Keyword-Based Search

Qi Su
Stanford University
qisu@cs.stanford.edu

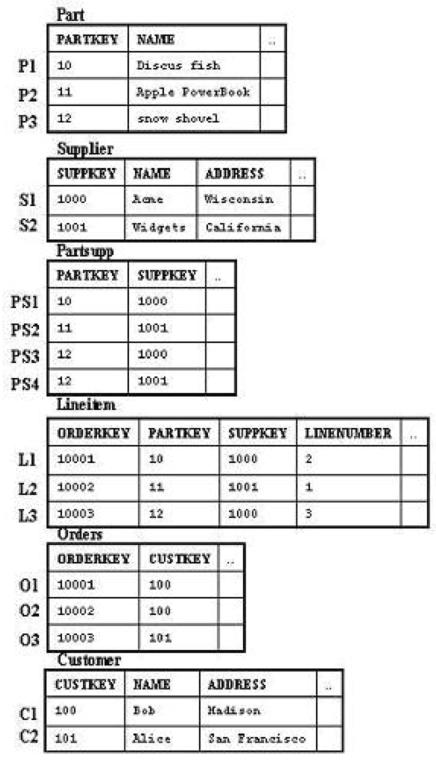
Jennifer Widom
Stanford University
widom@cs.stanford.edu

Abstract

Information Retrieval systems such as web search engines offer convenient keyword-based search interfaces. In contrast, relational database systems require the user to learn SQL and to know the schema of the underlying data even to pose simple searches. We propose an architecture that supports highly efficient keyword-based search over relational databases: A relational database is "crawled" in advance, text-indexing virtual documents that correspond to interconnected database content. At query time, the text index supports keyword-based searches with interactive response, identifying database objects corresponding to the virtual documents matching the query. Our system, EKSO, creates virtual documents from joining relational tuples and uses the DB2 Net Search Extender for indexing and keyword-search processing. Experimental results show that index size is manageable and database updates (which are propagated incrementally as recomputed virtual documents to the text index) do not significantly hinder query performance. We also present a user study confirming the superiority of keyword-based search over SQL for a range of database retrieval tasks.

1. Introduction

Relational Database Management Systems provide a convenient data model and versatile query capabilities over structured data. However, casual users must learn SQL and know the schema of the underlying data even to pose simple searches. For example, suppose we have a customer order database, shown in Figure 1, which uses the TPC-H [4] schema in Figure 2. We wish to search for Bob's purchases related to his aquarium fish hobby. To answer this query, we must know to join the Customer, Lineitem, Orders, and Part relations on the appropriate attributes, and we must know which attributes to constrain for "Bob" and "fish". The following SQL query is formulated:



Part

	PARTKEY	NAME	..
P1	10	Discus fish	
P2	11	Apple PowerBook	
P3	12	snow shovel	

Supplier

	SUPPKEY	NAME	ADDRESS	..
S1	1000	Acme	Wisconsin	
S2	1001	Widgets	California	

Partsupp

	PARTKEY	SUPPKEY	..
PS1	10	1000	
PS2	11	1001	
PS3	12	1000	
PS4	12	1001	

Lineitem

	ORDERKEY	PARTKEY	SUPPKEY	LINENUMBER	..
L1	10001	10	1000	2	
L2	10002	11	1001	1	
L3	10003	12	1000	3	

Orders

	ORDERKEY	CUSTKEY	..
O1	10001	100	
O2	10002	100	
O3	10003	101	

Customer

	CUSTKEY	NAME	ADDRESS	..
C1	100	Bob	Madison	
C2	101	Alice	San Francisco	

Figure 1. Example database instance

```
SELECT c.custkey, o.orderkey, p.partkey
FROM Customer c, Lineitem l, Orders o, Part p
WHERE c.custkey = o.custkey AND l.partkey = p.partkey
AND o.orderkey = l.orderkey AND c.name LIKE '%Bob%'
AND p.name LIKE '%fish%'
```

In our example database instance, this query returns a single tuple $\langle custkey, orderkey, partkey\rangle$ of $\langle 100, 10001, 10\rangle$. In contrast, keyword-based search, exemplified by web search engines, offers an intuitive interface for searching textual content that requires little knowledge about the underlying data. In this paradigm, for the above example a user should be able to enter the keywords "Bob fish" and find the relevant information from the results.

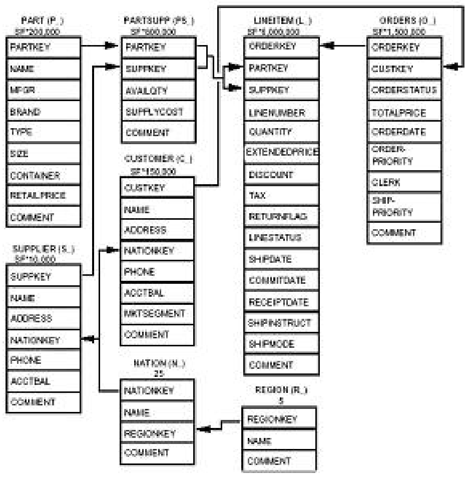


Figure 2. TPCH Schema

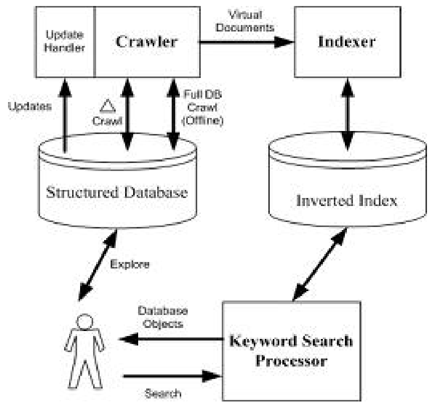


Figure 3. Architecture

We have developed a system, EKSO (for Efficient Keyword Search through Offline Indexing), that supports highly efficient keyword-based search on relational databases. For the above example, on query "Bob fish" EKSO returns a result equivalent to the SQL query result. EKSO also supports much more complex searches over complex data. Our novel use of DB2's Net Search Extender (NSE) [7, 13] enabled us to build a powerful system that is extensible, and that can incorporate future text indexing and search features in NSE without modification. Our system is the first we know of for expressive keyword-based search over interconnected relational database content that operates within the database engine and performs full offline indexing for highly efficient searches.

EKSO is an instance of a general architecture we propose in this paper, depicted in Figure 3 and consisting of three main components:

- The **Crawler** traverses the entire database offline, constructing virtual documents from database content. The Crawler can construct textual virtual documents in any fashion from the database (typically based on interconnected tuples from multiple relations if the database is relational), as long as each virtual document is associated with some logical or physical database object that makes sense when returned to the user. The Crawler includes an Update Handler that computes changes to virtual documents when the underlying database is updated.
- Virtual documents are sent to the **Indexer**, which builds an inverted index over the virtual documents in the usual Information Retrieval (IR) style. When the Crawler identifies new or modified virtual documents as a result of database updates, the Indexer refreshes the inverted index incrementally.
- At search time, the **Keyword Search Processor** takes a keyword query and probes the inverted index (again in an IR style) to find the virtual documents satisfying the query. The database objects associated with these documents are returned to the user. The user may then choose to explore the database or refine the query.

The key features and advantages of our approach, and the contributions of this paper, are summarized as follows:

- Most previous approaches to keyword-based search in structured databases (reviewed in Section 2) perform a significant amount of database computation at search time. Our offline indexing approach does all significant work in advance so it can provide interactive answers. We show in our performance study that the potential obstacles to this approach-index construction and update handling-are manageable. Index construction performance depends primarily on DB2, but can easily be parallelized. Updates do not have a dramatic impact on query performance, although for databases with very high update rate, one of the existing online approaches may be more appropriate.
- Our approach is extensible. We construct virtual documents through user-defined functions (UDFs) that in turn execute SQL queries. Currently our virtual documents are formed by concatenating text fields in interconnected relational tuples. However, without changing the infrastructure we could plug in a different UDF to construct virtual documents differently. For example, we could produce XML or some other textual format as virtual documents, possibly incorporating schema information. If the text search engine provides XML search capabilities such as *XPath* [3], our system automatically inherits those capabilities.
- We exploit existing relational database extensions for text indexing and search that is highly expressive and

extensible in its treatment of database content. By using DB2's Net Search Extender as the back end, we completely avoid reimplementing basic IR capabilities. Furthermore, our method for exploiting the text extension ensures that all features are automatically available in our search tool, including stemming, stopword elimination, case-folding, Boolean queries, proximity queries, relevance-ranked results, wildcards, fuzzy searches, and natural language searches [7].

- The premise of our work is that keyword-based search is more intuitive and convenient than SQL for many query tasks over relational databases. To verify this assumption, we conducted the only study we are aware of that quantifies the relative performance of SQL queries versus keyword searches on a variety of query tasks.

1.1. Outline of Paper

We provide a thorough survey of related work in Section 2. Section 3 is the core of the paper. It describes the EKSO system, including functionality, algorithms, and system implementation details. Section 4 presents our performance study of the EKSO system, while Section 5 presents our user study demonstrating the advantages of keyword-based searches over SQL queries. We conclude and suggest future directions in Section 6.

2. Related Work

An early paper [12] proposes implementing abstract data types, user-defined operators, and specialized indexes for textual databasesa precursor to current relational database vendors' approaches to integrating full-text search functionality. IBM DB2 [13], Microsoft SQL Server [9], Oracle [6], PostgreSQL, and MySQL all provide text search engine extensions that are tightly coupled with the database engine. However, in all cases each text index is defined over a single column. Using this feature alone to do meaningful keyword search over an interconnected database would require merging the results from many column text indexes. If our interconnected tuples contain N text attributes, and our keyword query contains M conjunctive terms, then in addition to joining relations to generate the interconnected tuples (as we do offline; see Section 3.1), we must invoke the text search engine up to NM times to match the query keywords to the text columns.

Three systems share the concept of crawling databases to build external indexes. Verity [15] crawls the content of relational databases and builds an external text index for keyword searches, as well as external auxiliary indexes to enable parametric searches. This approach is similar to ours in terms of the index-all-data-offline paradigm. However, by leveraging the database vendor's integrated text search engine tool, we avoid transporting data outside the database for external processing, which can be expensive and difficult to synchronize, and we inherit all DBMS features automatically.

DataSpot [5] extracts database content and builds an external, graph-based representation called a hyperbase to support keyword search. Graph nodes represent data objects such as relations, tuples, and attribute values. Query answers are connected subgraphs of the hyperbase whose nodes contain all of the query keywords. Similar to our concept of a *root tuple* (Section 3.1.1), each answer is represented by a node associated with a tuple in the database, from which the user can navigate to related tuples. As with Verity, DataSpot requires transporting content to be indexed outside the database.

DbSurfer [19] indexes the textual content of each relational tuple as a virtual web page. For querying and navigation, the database foreign-key constraints between tuples are treated as hyperlinks between virtual web pages. Given a keyword query, the system computes a ranked set of virtual web pages that match at least one keyword. Then a best-first expansion algorithm finds a ranked set of navigation paths originating from the starting web pages. The navigation path concept is similar to our definition of a text object (Section 3.1.2). However, even though the web pages are indexed offline, significant query time computation is required to expand the starting web page set to find navigation paths satisfying the full query, work that we perform offline.

Three systems, DBXplorer, BANKS, and DISCOVER, share a similar approach: At query time, given a set of keywords, first find tuples in each relation that contain at least one of the keywords, usually using auxiliary indexes. Then use graph-based approaches to find tuples among those from the previous step that can be joined together, such that the joined tuple contains all keywords in the query. All three systems use foreign-key relationships as edges in the graph, and point out that their approach could be extended to more general join conditions. We discuss the three systems in more detail.

Given a set of keywords, DBXplorer [1] first finds the set of tables that contain at least one of the keywords. Then using the undirected schema graph (where each node is a relation and each edge is a foreign-key relationship), a set of subgraphs is enumerated, such that each subgraph represents a joining of relations where the result contains rows that potentially contain all of the query keywords. For each candidate join subgraph generated, a SQL statement is executed to join the specified relations and select rows that do contain all of the keywords. The system is evaluated on TPC-H (100MB to 500MB) and three internal Microsoft

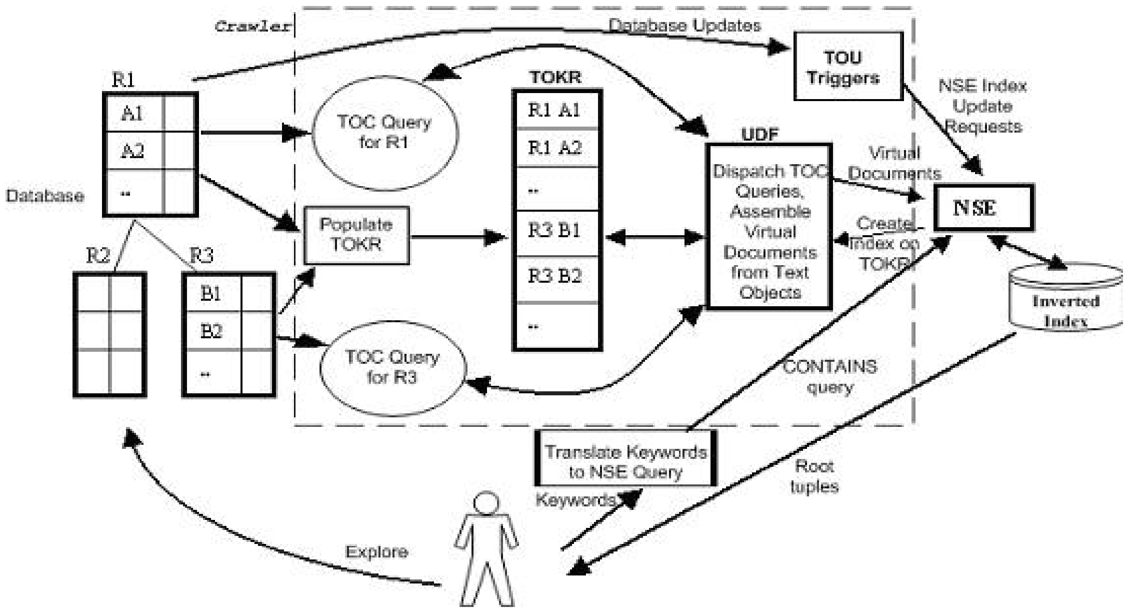


Figure 4. EKSO System

databases (130MB to 375MB, from 19 tables to 84 tables). The paper focuses on matching keywords to an entire attribute value. It briefly discusses extending the system to work with subattribute granularity matches and to provide more sophisticated search options.

BANKS [2] uses a data graph where each tuple is a node and each foreign-key relationship between tuples is represented as a bidirectional edge. The keyword query result is a connected subgraph of tuples, where each keyword is found in at least one tuple node. Dijkstra's algorithm is used to find the shortest path such that the nodes covered collectively contain all query keywords. The system can rank the results based on the sum of the node relevance scores and edge weights.

DISCOVER [11] uses schema graph information to enumerate minimum joining networks, which are sequences of tuples that can be joined together (based on foreign-key relationships) into a single tuple that contains all of the keywords. The algorithm models a tuple-set graph. Execution plans for the set of joining networks are generated and optimized to share intermediate results, then translated into SQL statements to build the join tuples that contain all of the query keywords. The performance evaluation focuses primarily on two aspects, efficiency of pruning irrelevant joining networks, and comparison of several algorithms for deciding what intermediate results to build and share. Execution time increases significantly with the number of candidate joining networks or the number of keywords in the query. [10] extends DISCOVER to perform relevance-ranked and top-k keyword queries.

An alternate approach, taken in [14, 16], translates keyword searches to SQL based on the schema, but does not focus on efficient implementation techniques. [8] describes a system for keyword and proximity search in graph-structured databases based on shortest-path computations, focusing on a combination of offline and online computation.

While a direct empirical comparison between our system and some of the other approaches mentioned in this section would be very interesting, the systems are not publicly available, and any effort to implement them well enough for a fair comparison would be prohibitive.

3. The EKSO System

The system we have developed, EKSO, is an instantiation of the general architecture we propose for keyword-based search over structured databases that was shown in Figure 3. Figure 4 shows details of the EKSO system, which will be described as this section proceeds.

3.1. Text Objects and Virtual Documents

Recall from Section 1 that our goal is to perform a full crawl of the database, generating virtual documents from database content that define the granularity of text search and retrieval in the system. In EKSO we separate two concepts: *Text objects* are generated from interconnected tuples based on joining relational tables. *Virtual documents* are generated from text objects by concatenating their string attributes. This separation has at least two advantages: First,

```
input  : Schema graph $\mathcal{G}$ where nodes are relations and edges are
         foreign key constraints and set of root relations $\mathcal{RS}$
output : Set of text objects $\mathcal{T}$
$\mathcal{T} \leftarrow \{\}$;
foreach root relation $\mathcal{R}$ in $\mathcal{RS}$ do
    foreach tuple $r$ in $\mathcal{R}$ do
        text object $t \leftarrow \{r\}$ ;
        start at $\mathcal{R}$, do breadth-first traversal on $\mathcal{G}$ ;
        foreach relation $\mathcal{R}$' traversed do
            add to $t$ the set of $\mathcal{R}$' tuples that foreign-key join
            with existing tuples in $t$ ;
        add $t$ to $\mathcal{T}$ ;
return $\mathcal{T}$ ;
```
Figure 5: Text Object Construction (TOC) algorithm

Example: Consider the text object construction process for one root tuple $P1$ from relation Part in the database instance in Figure 1. Starting with tuple $P1$, in the first step of the traversal we select Partsupp tuple $PS1$ and Lineitem tuple $L1$, which both join with $P1$ on the partkey attribute. Next we select Supplier tuple $S1$, which joins with $PS1$ on suppkey, and Orders tuple $O1$, which joins with $L1$ on orderkey. Then we select Customer tuple $C1$, which joins with $O1$ on custkey. The text object for $P1$ consists of the set of tuples $\{P1, PS1, L1, S1, O1, C1\}$.

Figure 6: Text Object Construction example

we can use SQL queries, which are optimized automatically by the DBMS, to generate text objects. Secondly, we can independently plug in different functionality for generating text objects from the database, or for generating virtual documents from text objects.

In EKSO we define a text object to be the set of tuples in the database that are related to a given *root tuple*. When a virtual document D is part of the answer to a keyword query, the system returns an identifier for the root tuple associated with the text object from which D was generated.

3.1.1 Root Relations and Tuples

Root tuples are defined as all of the tuples in *root relations*. Root relations can be designated automatically or set by the database administrator. Designating a relation R as a root relation creates text objects and virtual documents associated with each tuple of R. In EKSO as a default we select as root relations all "entity relations," i.e., all relations whose primary key is not composed entirely of foreign-keys.

3.1.2 Text Objects

We define the text object for a particular root tuple r as the set of all tuples in the database that join with r through a sequence of zero or more foreign-key joins. We do not backtrack, i.e., the sequence of foreign-key joins joins with any relation at most once. See example in Figure 6.

Figure 5 outlines the text object construction (TOC) algorithm. To construct the text object for a root tuple r of root relation R, we start with r. Logically we perform a breadth-first traversal of the schema graph. For all neighbor relations of R, we select those tuples that join with r, based on foreign-key constraints. Next we join those tuples with relations two away from the root relation R, and so on, until we have traversed all relations reachable from R in the schema graph. In the schema graph traversal, we treat foreign key constraints as bidirectional. We mark each relation during the breadth-first traversal to avoid backtracking and cycles in the schema graph.

The algorithm as described is not guaranteed to cover the entire database when creating text objects, unless certain conditions hold. We must have a set of root relations such that all relations in the database are reachable from some root relation via the schema graph. Otherwise we may have "dangling relations", in which case we could use a second pass to create additional text objects from the dangling content. Note that because we follow foreign key edges, there will be no dangling tuples within connected relations.

Alternate definitions could be used for text objects from a given root tuple. For example, we could follow equijoins as well as foreign-key joins (with a second pass to handle dangling tuples), and we could backtrack. In the Figure 6 example, the text object for Part tuple P1 includes the suppliers who supply P1, as well as customers who have ordered P1. We could backtrack to include other parts supplied by those suppliers who supply P1, as well as other parts ordered by those customers who ordered P1.

3.1.3 Virtual Documents

After a text object has been computed, we concatenate all textual attributes of all tuples in the text object to form the virtual document. The order of the tuples selected during traversal is nondeterministic due to SQL semantics, and in the current system we are not concerned about the concatenation order. However, if we extend the system to richer virtual documents, as we will discuss next, we may need to be careful about concatenation order. We can also include nontextual fields, such as numbers and dates, by simply using their string representation. As an example, we concatenate the textual attributes of the six tuples that comprise the text object for root tuple P1 from the Figure 6 example. For one possible ordering the virtual document would look like: `Discus fish Acme Wisconsin Bob Madison.`

Note that for now we have chosen one particular definition of text objects and virtual documents. We can easily plug in different definitions. For example, we might generate richer structures in virtual documents. If the Indexer and the Keyword Search Processor supported XML content, we could generate an XML fragment that captures database structure and uses tags to encode attribute and relation names. Such virtual documents enable XPath [3] or other XML-based queries.

3.2. Crawler

The Crawler is responsible for traversing a structured database to produce virtual documents to be indexed. In Figure 4, the three relations on the left-hand side labeled R1, R2, and R3 correspond to the Structured Database in Figure 3. The Crawler in Figure 3 maps to the bulk of the system diagram in Figure 4, enclosed in the dashed box. It centers on a user-defined function (UDF) implemented in DB2 that takes as input an identifier for a root tuple, invokes a Text Object Construction (TOC) query, and constructs and returns a virtual document. The TOC queries are composed in advance following the TOC algorithm from Figure 5. There is one such query for each root relation. With knowledge of the schema, SQL WITH queries parameterized by each root relation's primary key values are used.

Before indexing, a special table called *Text Object Key Relation(TOKR)* is populated with keys that individually identify all of the root relations and tuples in those relations. At indexing time, the Crawler UDF decodes each key to find the root relation that the tuple belongs to, then looks up the corresponding TOC query, parameterizes it with the root tuple identifier values, and executes it. The query result is the text object, whose textual attributes are concatenated into a Binary Large Object (BLOB). At this point, the UDF discards the text object and returns the BLOB as the virtual document.

3.3. Indexer

The Indexer is the box labeled NSE in the right-hand portion of Figure 4. It is managed by the DB2 Net Search Extender (NSE). NSE is a full-featured text search engine integrated with the IBM DB2 DBMS product. NSE supports the creation of text indexes on individual columns, and can index the output of applying a user-defined function to a column, which is the feature we exploit. We create an NSE-managed IR-style inverted index over the virtual documents returned by the Crawler UDF described in the previous section. NSE indexes all of the virtual documents, associating each one with its corresponding text object key.

3.4. Keyword Search Processor

In Figure 4, the Keyword Search Processor from Figure 3 is the box labeled ”Translate Keywords to NSE Query”. A user's keyword-based query is translated to an NSE invocation in the form of a SQL query invoking the special CONTAINS function. For example, after indexing our database instance from Figure 1, the conjunctive keyword query ”Bob fish” is translated into the following SQL query:

```
SELECT key FROM Tokr WHERE CONTAINS(key,'"Bob"&
"Fish"')=1
```

The CONTAINS function automatically invokes the NSE search engine, which probes its inverted index and returns text object keys as query results, which are then returned to the user. The user may choose to explore the database from these keys, or further refine the search. We describe our user interface that enables these activities in Section 3.6. Currently user exploration begins from scratch. To speed up user traversal we could consider caching traversal path from text object construction.

By building on commercial DBMS text search features, we automatically inherit all of its capabilities, current as well as future. For example, EKSO automatically delivers ranked search results based on NSE's built-in ranking functions.

3.5. Update Handler

The Update Handler that is a part of the Crawler component in Figure 3 corresponds to the box labeled ”TOU Triggers” at the top right of the system diagram in Figure 4. When one or more tuples in the underlying database are modified (inserted, deleted, or updated), some text objects

```
input  : Schema graph $\mathcal{G}$ where nodes are relations and edges are
         foreign key constraints, set of root relations $\mathcal{RS}$, and an
         inserted, deleted, or updated tuple $r$ from Relation $\mathcal{R}$
output : Root tuples $\mathcal{RT}$ whose text objects and virtual
         documents need to be recomputed

$\mathcal{RT} \leftarrow \{\}$;
$\mathcal{RT}$' $\leftarrow \{r.old, r.new\}$;
start at $\mathcal{R}$, do breadth-first traversal on $\mathcal{G}$ ;
foreach relation $\mathcal{R}$' traversed do
    $\mathcal{D} \leftarrow$ set of tuples from $\mathcal{R}$' that foreign-key join with existing
    tuples in $\mathcal{RT}$';
    add $\mathcal{D}$ to $\mathcal{RT}$';
    if $\mathcal{R}$' in $\mathcal{RS}$ then
        add $\mathcal{D}$ to $\mathcal{RT}$ ;
return $\mathcal{RT}$ ;
```

Figure 7: Text Object Update (TOU) algorithm

	Root tuple r	Non-root tuple r
r inserted	Index TOC(r), Modify TOU(r)	Modify TOU(r)
r deleted	Delete index entries for r, Modify TOU(r)	Modify TOU(r)
r updated	Delete index entries for $r.old$, Modify TOU(r) Index TOC($r.new$),	Modify TOU(r)

Table 1. Text Object Update action

and virtual documents may need to be created, deleted, or updated. For a given modified tuple r, the first step is to compute the *delta set* of root tuples whose text object and virtual document must be recomputed. We call this computation *Text Object Update (TOU)* and we denote the delta set of root tuples as TOU(r). Given a modified tuple r, TOU(r) is the union of the following two sets: the set of existing root tuples whose text object includes the old version of r and the set of existing root tuples whose text object should now include the new version of r.

If tuple r is inserted, the old-version set is empty. If tuple r is deleted, the new-version set is empty. Once the delta set TOU(r) is computed, for every root tuple r' in TOU(r), we repeat the crawling and indexing procedure described earlier: The Crawler computes the text object and virtual document of r', then the Indexer removes any existing index entries of r' and indexes the recomputed virtual document.

In Figure 7 we present the TOU algorithm to compute the delta set of root tuples whose text objects and virtual documents must be recomputed as a result of modifications to an arbitrary tuple. Note the TOU algorithm is a dual of the TOC algorithm, since the TOU algorithm must find affected root tuples based on the text object semantics. Hence if the text object definition were to change (e.g., to permit backtracking), then both the TOC and the TOU algorithms must change accordingly. Like the TOC algorithm, the TOU algorithm can be implemented as a SQL WITH query composed offline using schema information. There is one TOU query per relation in the database.

Once the delta set of root tuples is identified, our update actions depend on whether the modified tuple r is itself a root tuple, and whether the original modification to r was an insert, delete, or update. Table 1 enumerates all the cases. $r.old$ denotes the old version of a modified tuple r, i.e., a tuple being deleted, or a tuple before it is updated. $r.new$ denotes the new version of a modified tuple r, i.e., a tuple being inserted, or a tuple after it has been updated. ”Modify TOU(r)” denotes asking the Crawler to recompute the text objects and virtual documents of each root tuple in the delta set TOU(r), and asking the Indexer to update its index with the recomputed virtual documents. When we insert a new root tuple r we must also ask the Crawler to execute the TOC algorithm for r and ask the Indexer to index the virtual document generated by the Crawler, which we denote by ”Index TOC(r)”. When we delete a root tuple, we must have the Indexer remove index entries associated with the deleted root tuple.

The Update Handler is implemented using triggers. We create three TOU triggers (insert, delete and update) on each database relation. The main trigger action for all three triggers is an insert statement that executes the TOU query to find the set of root tuples whose virtual document is affected by changes to a given tuple, and then inserts into the NSE update log table requests to update the NSE index entries for these root tuples. At the next NSE index update (either a manual NSE index refresh request or automatic periodic index refresh), NSE automatically invokes the Crawler UDF to reconstruct the requested virtual documents, which are used to rebuild the inverted index.

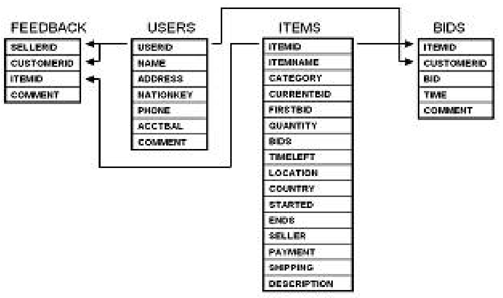


Figure 8. Auction dataset schema

In the case of the insert trigger on a root relation, a second trigger action inserts a corresponding tuple into the TOKR, which automatically creates an NSE update request that a new virtual document be generated and indexed. For the delete trigger on a root relation, a second trigger action deletes the corresponding TOKR tuple, thus generating an NSE update request that the appropriate entries in the NSE index be removed.

3.6. User Interface

We have developed an Application Programming Interface (API) to our system that enables an administrator to quickly and easily deploy a customized web-based front end to the search system for any database schema. Recall that our approach yields search results that are simply identifiers for the root tuples of the virtual documents matching the search. Thus we have also developed a servlet-based web user interface that supports navigation starting from the root tuples returned by the keyword search, and following the same breadth-first traversal our TOC algorithm follows. In Section 5, we present a user study comparing the effectiveness of keyword versus SQL searches through our interface and system.

4. System Evaluation

Our system is implemented on a PC with dual Pentium III 800MHz processors and 1GB of RAM, running Windows 2000, IBM DB2 UDB version 8.1, and DB2 Net Search Extender version 8.1.

Relation	Data Source	Root Reln	#Tuples	Bulkload File Size
ITEMS	Ebay	Yes	500,000	795MB
USERS	TPC dbgen	Yes	26,913	4MB
FEEDBACK	TPC dbgen	No	500,000	76MB
BIDS	TPC dbgen	No	1,000,000	54MB

Table 2. Auction dataset characteristics

Dataset	Bulkload File Size	DB Size	Root Tuples	NSE Index Size
TPCH500	500MB	597MB	930,000	4.16GB
Auction	929MB	1128MB	526,913	2.76GB

Table 3. Dataset space utilization

4.1. Datasets

We evaluate our text indexing and search system on two datasets, a 500MB TPC-H database, and a 929MB Auction database. The TPCH500 dataset follows the complete TPC-H schema in Figure 2 and is generated using the official TPC-H data generator (DBGEN) [4]. Schema information including relation names, attribute names, and join conditions are provided to our TOC Query construction application. We designate Customer, Orders, Part, and Supplier as the root relations. The system default would also include Region and Nation, but they are of too coarse a granularity to be useful to users. The four parameterized TOC Queries for the four root relations constitute our workload at indexing time, since each UDF invocation executes one of these queries and concatenates the resulting tuples. We optimized the indexing process by submitting this four-query workload to the DB2 Design Advisor, which suggested indexes to build on the database tables.

The Auction dataset consists of a mixture of primarily real-world data and some synthetic data. It models a database for an online auction service and the schema is shown in Figure 8. Whereas the TPC-H schema has many relatively short textual fields and is fairly complex with many join relationships, the Auction dataset contains a few long textual fields and is very simple in terms of join relationships. The Items table is populated with 500,000 Ebay auction items crawled in 2001, provided courtesy of the University of Wisconsin database group. The Users table is populated from a subset of the Supplier table in our TPCH500 dataset. The Feedback and Bids tables are synthetically generated, with the comment attributes populated with pseudo-random text output from the TPC DBGEN utility. We designate the two entity relations, Items and Users, as the root relations.

Table 2 summarizes the Auction dataset. The two TOC Queries were presented to DB2 Design Advisor for optimization. Table 3 presents the space utilization for the two datasets. Database sizes are obtained using the DB2 table size estimation utility and does not include database indices. The TPCH500 dataset contains 930,000 root tuples from the Customer, Orders, Part, and Supplier relations, representing 930,000 TOC Queries executed at indexing time. The Auction dataset contains 526,913 root tuples from the Items and Users relations for 526,913 TOC Queries at indexing time.

Our system executes several TOC Queries per second on average, though performance could improve with various levels of tuning. Also, the TOC query workload is well suited for parallelization.

	Full Results			Top 100 Ranked	
Query Length (words)	Query Time (sec)	95% Conf Interval (sec)	Result Size (tuples)	Query Time (sec)	95% Conf Interval (sec)
2	24.05	0.55	157970	3.89	0.29
3	21.82	0.44	82994	3.99	0.24
4	21.48	0.30	51996	4.69	0.26

Table 4. TPCH500 query performance

	Full Results			Top 100 Ranked	
Query Length (words)	Query Time (sec)	95% Conf Interval (sec)	Result Size (tuples)	Query Time (sec)	95% Conf Interval (sec)
2	6.02	0.43	25604	2.21	0.11
3	3.46	0.20	6851	1.64	0.07
4	2.29	0.36	3002	1.38	0.15

Table 5. Auction query performance

4.2. Keyword Searches

We evaluate search performance by submitting conjunctive keyword queries of length 2, 3, and 4 words. We evaluate returning full result set and the top 100 ranked results. In each trial, we generate 50 queries by randomly choosing words from the appropriate corpus with replacement. The reported time for a trial is the average of the 50 query execution times (prepare plus execute plus fetch time) reported by DB2's query performance monitoring utility, db2batch. We perform 20 trials for each type of query (number of keywords plus full or top-100 results) to report average query time with a 95% confidence interval.

The TPC-H dataset is generated by the TPC data generator which produces text attributes such as the comment field from a probability distribution file. Our TPC-H keyword queries are generated from the set of about 300 words that are used to generate the comment field.

In the Auction dataset, keyword queries are generated from the description field of the auction items table, which

Update Rate (tuples/sec)	0	.125	.25	.5	.75	1	2	4
Avg Query Wkld Time (sec)	.933	1.29	1.33	1.34	1.30	1.28	1.23	1.25

Table 6. Update performance

came from real Ebay auction pages. We exclude words of length 3 letters or less. Query keywords are chosen from a bag of 51 million words.

4.3. Query Performance

Table 4 shows query performance on the TPCH500 dataset. The performance of queries returning the full result set are not interactive, at 21-24 seconds, due to the size of the result (50K-160K tuples). Typically users are only interested in the top few results of a keyword query. Our top-100 ranked query performance is interactive at under 4.7 seconds. TPC-H is not an ideal dataset for keyword search because the text attributes are generated from a very limited corpus, producing large results for experiments. Auction queries return many fewer results. Table 5 shows that query times for both the full results and the ranked queries are interactive for the Auction dataset, at under 6.1 seconds and under 2.3 seconds, respectively.

4.4. Update Performance

We generate a 500MB database by randomly selecting half of the tuples from a 1GB TPC-H dataset. From the remaining tuples, we pick 10,000 tuples to form 100 batches of 100 tuples each. Within each batch, the proportion of tuples from each relation reflects the relation's proportion in the TPC-H dataset specification. The update workload consists of a sequence of inserting 100 batches of 100 tuples each. We pause for some time between consecutive insert batches in order to achieve a target insert rate (tuples per second). We measure query performance at insert rates of 0, 0.125, 0.25, 0.5, 0.75, 1, 2, and 4 tuples per second. The low insert rates are due to the fact that NSE index update saturates at 0.75 tuple inserts per second, and database update saturates at about 10 tuple inserts per second.

Query performance is measured by a client that executes keyword queries concurrently with the update workload. The query client repeatedly executes a workload of 9 randomly generated TPCH500 keyword queries, with a one-second pause between workload executions. We report the average time to execute one query workload during the period that the update workload of 100 batches are being inserted into the database, which means we measure thousands of query workload executions.

Task ID	Avg Query Time (sec)		Avg # Queries Submitted	
	SQL	Keyword	SQL	Keyword
1	193	54	1.43	1
2	332	76	3.75	1.43
3	206	49	2.29	1.13
4	76	58	2	1.25
5	228	160	1.88	3
6	149	96	1.38	1.5
7	99	126	1.5	2

Table 7. User study results

Our results are shown in Table 6. Without inserts, our average query workload time is 0.933 seconds. For the seven insert workloads, the average query workload time ranges from 1.23 to 1.34 seconds, without obvious correlation between query time and update load. Our hypothesis is that the variation from 1.23 to 1.34 seconds probably reflects noise in normal system fluctuation, while the jump from 0.933 to about 1.3 seconds is indicative of the actual overhead from the update workload. At the insert rate of 0.75 tuples per second, the background NSE index update process becomes saturated. See technical report [18] for more detailed discussion.

5. User Study

We conducted a study involving 17 doctoral students and faculty in the Stanford Database Group to quantify the relative effectiveness of keyword and SQL interfaces in accessing content in relational databases. We used a movie database [20] consisting of three relations, Movies, Casts, Actors, and a view, Movie_actor. Using our API described in Section 3.6, we deployed a servlet-based web interface using Apache Tomcat on the same PC used in our system evaluation. See technical report [18] for more details on the user study.

Two classes of questions were formulated. The first class, questions 1, 2, and 3, clearly favors keyword searches. For example, for question 1, the 3-keyword query "Burton Taylor Cinecitta" returns a single movie tuple for the movie Cleopatra, which is the correct answer. On the other hand, questions 4-7 were more difficult with keyword search: The best keyword queries return many result tuples that the user must browse to find the answer. Furthermore, for questions 5 and 6 the user must submit two consecutive keyword queries to find the correct answer. In comparison, all 7 questions could be answered with a single SQL query.

Our results in Table 7 show that even for a group of knowledgeable SQL users, in most cases keyword search is more effective than SQL. Even for the category of questions that do not favor keyword search, our users answered three out of four questions faster using keyword search than using SQL.

For questions 1 through 6, on average the keyword search time is 48% of the SQL search time. For question 7, the keyword search time is 127% of the SQL time. Even though all questions could be answered with a single SQL query, the average number of SQL queries submitted column illustrates that SQL is error-prone (even with our schema-driven query formulation interface). Of course SQL is inherently more verbose than keyword search, which may somewhat bias the task completion times, but we believe the overall results are still meaningful.

6. Conclusion and Future Work

We presented a general architecture for supporting keyword-based search over structured databases, and an instantiation of the architecture in our fully implemented system EKSO. EKSO indexes interconnected textual content in relational databases, providing intuitive and highly efficient keyword search capabilities over this content. Our system trades storage space and offline indexing time to significantly reduce query time computation compared to previous approaches.

Experimental results show that index size is manageable and keyword query response time is interactive for typical queries. Update experiments demonstrate that our concurrent incremental update mechanism does not significantly hinder query performance. We also present a user study confirming the superiority of keyword-based search over SQL for a range of database retrieval tasks.

We have identified at least three main directions for future work:

- Indexing Performance: We plan to investigate better computation sharing when constructing text objects, as well as using parallelism to speed up indexing.
- Enhanced Search: By incorporating more structure into text objects, we can exploit (future) search capabilities for semistructured data. We will investigate how to generate text objects with structural information, focusing first on XML and exploiting work on publishing relational content as XML, e.g. [17].
- User Interface: We plan to refine and enhance our current search result presentation and navigation schemes.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *Proc. of ICDE*, 2002.

[2] Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proc. of ICDE*, 2002.

[3] World Wide Web Consortium. Xml path language version 2.0. `http://www.w3.org/TR/xpath20/`.

[4] Transaction Processing Council. `http://www.tpc.org/`.

[5] S. Dar, G. Entin, S. Geva, , and E. Palmon. Dtl's dataspot: Database exploration using plain languages. In *Proc. of VLDB*, 1998.

[6] P. Dixon. Basics of oracle text retrieval. *IEEE Data Engineering Bulletin*, 24(4), 2001.

[7] IBM DB2 Net Search Extender. `http://www-3.ibm.com/software/data/db2/extenders/netsearch/index.html`.

[8] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proc. of VLDB*, 1998.

[9] J. Hamilton and T. Nayak. Microsoft sql server full-text search. *IEEE Data Engineering Bulletin*, 24(4), 2001.

[10] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *Proc. of VLDB*, 2003.

[11] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *Proc. of VLDB*, 2002.

[12] C. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. In *Proc. of VLDB*, 1988.

[13] A. Maier and D. Simmen. Db2 optimization in support of full text search. *IEEE Data Engineering Bulletin*, 24(4), 2001.

[14] U. Masermann and G. Vossen. Design and implementation of a novel approach to keyword searching in relational databases. In *Proc. of ADBIS-DASFAA Symposium on Advances in Databases & Information Systems*, 2000.

[15] P. Raghavan. Structured and unstructured search in enterprises. *IEEE Data Engineering Bulletin*, 24(4), 2001.

[16] N. L. Sarda and A. Jain. Mragyati: A system for keyword-based searching in databases. `http://arxiv.org/abs/cs.DB/0110052`.

[17] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as xml documents. In *Proc. of VLDB*, 2000.

[18] Q. Su and J. Widom. Indexing relational database content offline for efficient keyword-based search. *Stanford University Technical Report*, 2003.

[19] R. Wheeldon, M. Levene, and K. Keenoy. Search and navigation in relational databases. `http://arxiv.org/abs/cs.DB/0307073`.

[20] G. Wiederhold. Movie database. `http://www-db.stanford.edu/pub/movies/doc.html`.