Keyword Proximity Search on XML Graphs

Vagelis Hristidis Yannis Papakonstantinou Andrey Balmin
Computer Science and Engineering Dept.
University of California, San Diego
{vagelis,yannis,abalmin}@cs.ucsd.edu

Abstract

XKeyword provides efficient keyword proximity queries on large XML graph databases. A query is simply a list of keywords and does not require any schema or query language knowledge for its formulation.

XKeyword is built on a relational database and, hence, can accommodate very large graphs. Query evaluation is optimized by using the graph's schema. In particular, XKeyword consists of two stages. In the preprocessing stage a set of keyword indices are built along with indexed path relations that describe particular patterns of paths in the graph. In the query processing stage plans are developed that use a near optimal set of path relations to efficiently locate the keyword query results. The results are presented graphically using the novel idea of interactive result graphs, which are populated on-demand according to the user's navigation and allow efficient information discovery. We provide theoretical and experimental points for the selection of the appropriate set of precomputed path relations. We also propose and experimentally evaluate algorithms to minimize the number of queries sent to the database to output the top-K results.

1 Introduction

XML and its labeled graph abstraction emerge as the data model of choice for representing semistructured self-describing data. Semistructured query languages (see [2] for a survey and [23] for the emerging XQuery standard) provide features, such as flexible path expressions, that allow one to query semistructured data, i.e., graph data that are not characterized by rigid structure. However, one still needs sufficient knowledge of the structure, role of the requested objects and XQuery in order to formulate a meaningful query. Keyword search does not present such requirements; it enables information discovery by providing a simple interface. It has been the most popular information discovery method since the user does not need to know either a query language or the structure of the underlying data.

The search engines available today provide keyword search on top of sets of documents. When a set of keywords is provided by the user the search engine returns all documents that are associated with these keywords. Typically, a set of keywords and a document are associated if the keywords are contained in the document. Their degree of associativity is often their distance from each other.

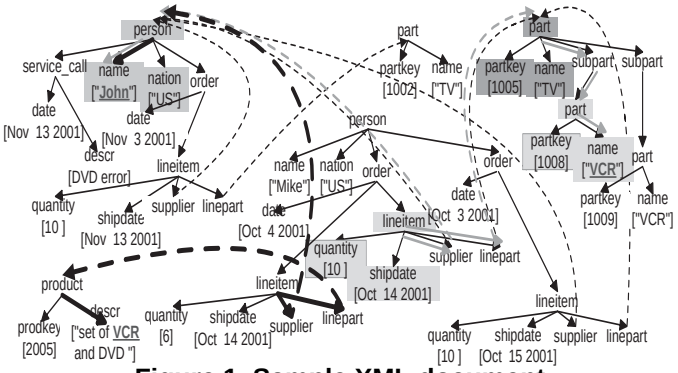


Figure 1. Sample XML document

XKeyword follows a recent generation of information retrieval systems that provide keyword proximity search [12, 13, 6, 3] to structured and semistructured databases. In particular, XKeyword provides keyword proximity search on XML data that are modeled as labeled graphs, where the edges correspond to the element-subelement relationship and to `IDREF` pointers. XKeyword differs from prior systems for proximity search on labeled graphs in that it assumes the existence of a schema, similar to the XML Schema standard [22], to which the graph conforms. The schema facilitates the presentation of the results and is also used in optimizing the performance of the system. Note that the end-user does not need to be aware of the schema.

A *keyword proximity query* is a set of keywords and the results are trees of XML fragments (called *target objects*) that contain all the keywords and are ranked according to their size. Trees of smaller sizes denote higher association between the keywords, which is generally true for reasonable schema designs. For example, consider the keyword query "John, VCR" on the graph of Figure 1. The first highlighted tree (thick edges) $name[John] \leftarrow person \leftarrow supplier \leftarrow lineitem \rightarrow linepart \rightarrow product \rightarrow descr[set\ of\ VCR\ and\ DVD]$ on the source XML graph of Figure 1 is a result of size 6. The second highlighted tree (gray arrows) $name[John] \leftarrow person \leftarrow supplier \leftarrow lineitem \rightarrow linepart \rightarrow part \rightarrow subpart \rightarrow part \rightarrow name[VCR]$ is a result of size 8. The first result is considered to be a "better" one by XKeyword (as well as by all the other keyword proximity search systems) since the shorter distance corresponds to the closer connection between "John" and "VCR" in the first solution, where the "VCR" is the product that "John" supplied, as opposed to

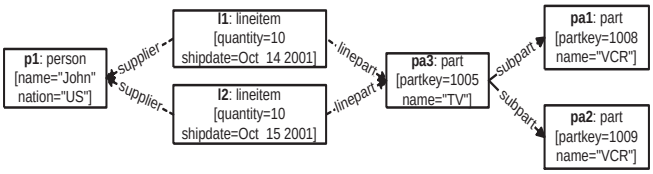


Figure 2. Multivalued dependencies in results

being a sub-part of another part supplied by "John". Notice that we allow edges to be followed in either direction.

The presentation of the results faces two key challenges that have not been addressed by prior systems. First, the results need to be semantically meaningful to the user. Towards this direction, XKeyword associates a minimal piece of information, called *target object*, to each node and displays the target objects instead of the nodes in the results. In the DBLP demo (Figure 4) XKeyword displays target object fields such as the paper title and conference along with a paper. In the TPC-H example, XKeyword adds some children nodes of the person, lineitem and part nodes (highlighted for the second result in Figure 1). For example, we display the *target object* $part[partkey[1005], name[TV]]$ in the place of the intermediate $part$ node. Target objects are designated by the system administrator who splits the schema graph in minimal self-contained information pieces (Figure 6), which we call *Target Schema Segments (TSS)* and correspond to the target objects presented to the user. Furthermore, the edges connecting the target objects in the presentation graph are annotated with their semantic description, which is defined on the TSS graph (Figure 6). For example the $part \rightarrow part$ edge is named "*subpart*".

The second challenge is to avoid overwhelming the user with a huge number of often trivial results, as is the case with DISCOVER [13] and DBXplorer [3][1]. Both of those systems present all trees that connect the keywords. In doing so they produce a large number of trees that contain the same pieces of information many times. For example, consider the keyword query "US, VCR" and the subgraph of the XML graph of Figure 1 shown in Figure 2. This XML fragment contains four results:

$N_1 : p_1 \leftarrow l_1 \rightarrow pa_3 \rightarrow pa_1, \quad N_2 : p_1 \leftarrow l_2 \rightarrow pa_3 \rightarrow pa_2,$
$N_3 : p_1 \leftarrow l_2 \rightarrow pa_3 \rightarrow pa_1, \quad N_4 : p_1 \leftarrow l_1 \rightarrow pa_3 \rightarrow pa_2$

The above results contain a form of redundancy similar to multivalued dependencies [20]: we can infer N_3 and N_4 from N_1 and N_2. In that sense, N_3 and N_4 are trivial, once N_1 and N_2 are given. Such trivial results penalize performance and overwhelm the user. XKeyword avoids producing "duplicate" results by employing a smart execution algorithm. On the presentation level it uses a *presentation graph* that comprises the complete set of nodes participating in result trees. At any point only a subset of the graph is shown (see Figure 3), as it is formulated by various navigation actions of the user. Initially the user sees one result tree r_0. By clicking on a node of interest the graph is expanded to display more nodes of the same type that belong to result trees that contain as many as possible of the other nodes of r_0. Towards this purpose we define a minimal expansion concept. For example, clicking on the $lineitem$ node of Figure 3 (a) displays all $lineitem$ nodes which are connected to the $person$ and $part$ in the initial tree, as shown

[1]Both systems work on relational databases, but the presentation challenges are similar.

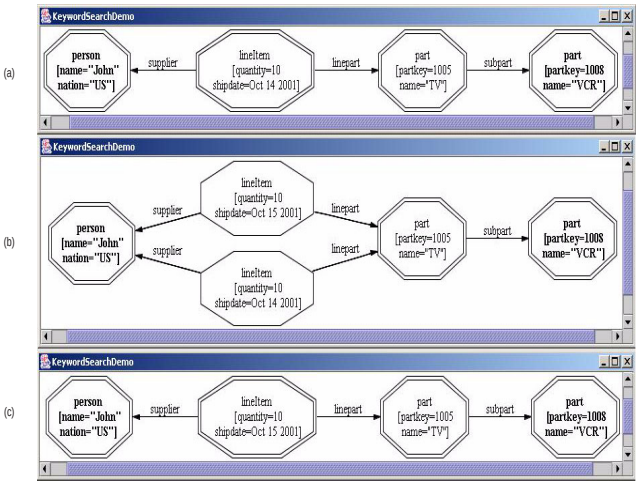


Figure 3. Presentation graph (expanded nodes have single outliner)

in Figure 3 (b).

Two key challenges arise on the way to providing fast response times. First, the XML data has to be stored efficiently to allow the fast discovery of connections between the elements that contain the keywords. We follow the architecture of multiple recent XML database systems and store the XML data in a relational database [7, 19, 10, 16, 8, 18, 5], which we tune to provide the needed indexing and clustering. Then XKeyword builds a set of *connection relations*, which precompute particular path and tree connections on the TSS graph. Connection relations are similar to path indices [9] since they facilitate fast traversal of the database, but also different because they can connect more than two objects and they store the actual path between a set of target objects, which is needed in the answer of the keyword query. A core problem is the choice of the set of connection relations that are precomputed.

Second, the cost of computing the full presentation graph is very high. Hence XKeyword uses an on-demand execution method, where the execution is guided according to the user's navigation. We present an algorithm that generates a minimal set of queries to the underlying database in response to the user's navigation.

XKeyword consists of two stages (Figure 7). In the preprocessing stage, the *master index* is created along with a set of connection relations. The master index is an inverted index that stores for each keyword k a list of elements that contain k. The most suitable *decomposition*, i.e., representation of the target object graph with a set of connection relations, is selected, given the performance and space requirements. We compared different decomposition strategies and found that in order to compute the top-1 result for each result schema, which is needed to construct the presentation graph, the most space effective decomposition is to create inlined fragments [5], i.e., fragments that do not contain multivalued dependencies. On the other hand, a combination of the inlined and the minimal decomposition, where a connection relation is generated for each edge of the schema graph, is more efficient for the on-demand expansion of the presentation graph.

In the query processing stage, XKeyword retrieves from

the master index the schema nodes, whose elements contain the keywords, and exploits the schema graph's information (in contrast to [12, 6]) to generate a complete and non-redundant set of connection trees (*candidate networks (CN)*) between them. Each CN may produce a number of answers to the keyword query, when evaluated on the XML graph. A presentation graph is generated for each CN, since they correspond to the different schemata of results. The *CN Generator* of XKeyword is an extension to XML databases of the CN Generator of DISCOVER [13]. We also present ways to improve the performance of the algorithm described in [13].

The CN's are passed to the query optimizer, which generates an *execution plan*. The key challenges of the optimizer are (a) to decide which connection relations to use to efficiently evaluate each CN and (b) to exploit the reusability opportunities of common subexpressions among the CN's. Both decisions, which are shown to be NP-complete, dramatically affect the performance as we show experimentally.

Finally, the results are presented to the user. XKeyword offers two presentation methods: displaying a presentation graph for each CN (Figure 4 (c)), or displaying a full list of results (Figure 4 (b)), where each result is a tree that contains every keyword exactly once. The former method offers a more compact and non-redundant representation, while the latter favors faster response times.

In summary, this paper makes a number of contributions in the area of keyword proximity search:

- We present keyword proximity search semantics, extended to capture our novel result presentation method, which prevents information overflow and allows the user to navigate in the result.
- We present an architecture and framework that allows for choosing which connections between objects will be precomputed. We present rules to avoid generating any *useless* connection relation, i.e., connection relations that are not efficient to evaluate any CN. We show how to bound the number of joins needed to output a solution.
- We address the on-demand performance requirement of the presentation approach and we compare and analyze different decomposition schemes with respect to it. We also present an algorithm that efficiently generates the full list of results by caching partial results and avoiding to recompute the common result portions and show experimentally that it is up to 80% faster than the naive approach used in [13] and [3].

XKeyword has been implemented (Figure 4) and a demo is available at http://www.db.ucsd.edu/XKeyword, which operates on the XML data of the DBLP database.

2 Related Work

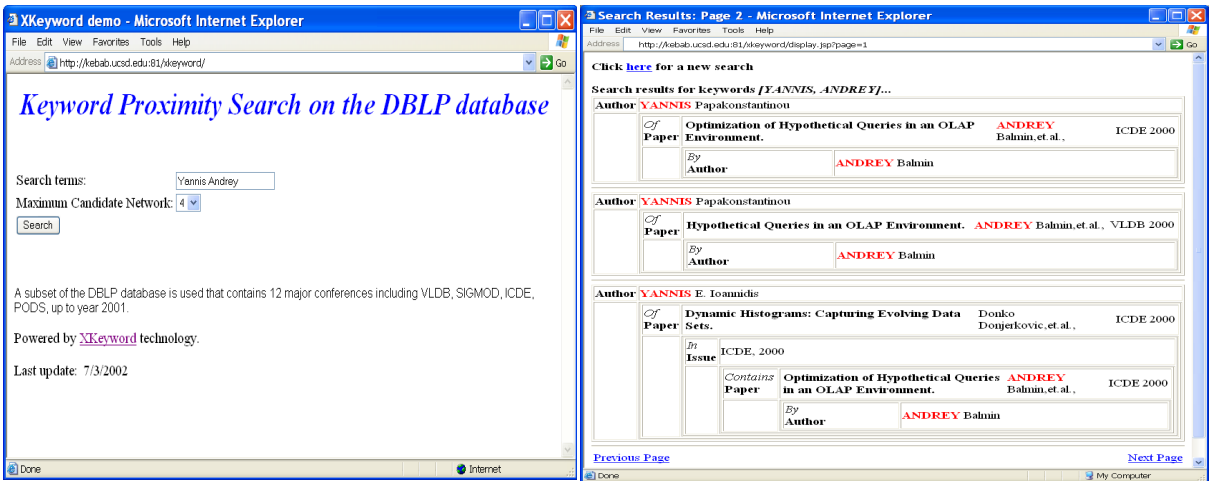
There is a number of proposals for less structured ways to query XML database by incorporating keyword search [11, 1] or by relaxing the semantics of the query language [15, 4]. However none of these works incorporates proximity search. Florescu et al. [11] propose an extension to XML query languages that enables keyword search at the granularity of XML elements, which helps novice users formulate queries. Another difference of this work from XKeyword is that it requires the user to specify the elements where the keywords are.

In [12] and [6], a database is viewed as a graph with objects/tuples as nodes and relationships as edges. Relationships are defined based on the properties of each application. For example an edge may denote a primary to foreign key relationship. In [12], the user query specifies two sets of objects, the $Find$ and the $Near$ objects. These objects may be generated from two corresponding sets of keywords. The system ranks the objects in $Find$ according to their distance from the objects in $Near$. An algorithm is presented that efficiently calculates these distances by building hub indices. In [6], answers to keyword queries are provided by searching for Steiner trees [17] that contain all keywords. Heuristics are used to approximate the Steiner tree problem. Two drawbacks of these approaches are that (a) they work on the graph of the data, which is huge and (b) the information provided by the database schema is ignored. In contrast, XKeyword (a) works on the relatively compact set of target objects connections (see Section 3) and (b) exploits the properties of the schema of the database. XKeyword also provides relatively scalable performance as the available space to store fragments (see Section 5) increases.

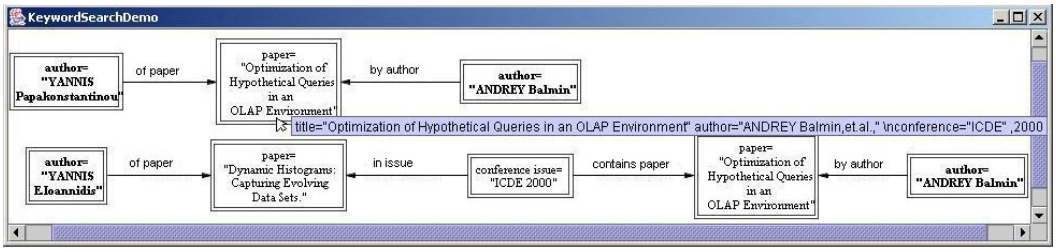
DISCOVER [13] and DBXplorer [3] work on top of a DBMS to facilitate keyword search in relational databases. They are middleware in the sense that they can operate as an additional layer on top of existing DBMS's. In contrast, XKeyword is a system dedicated to providing efficient keyword querying of XML databases, by using elaborate duplication and indexing techniques. XKeyword provides guarantees on the performance of the keyword queries, which is not possible for a middleware system. DISCOVER and DBXplorer do not consider building materialized views, which is the equivalent of redundant fragments in XKeyword. Furthermore, XKeyword adopts an elaborate presentation method using interactive graphs of results. In contrast, DISCOVER and DBXplorer output a list of results, including trivial ones. The inherent differences of XML from relational data are handled in XKeyword by introducing the notion of target object.

Both DISCOVER and XKeyword exploit reusability opportunities among the candidate networks, in contrast to DBXplorer. The candidate network generator of XKeyword is an extension of the candidate network generator of DISCOVER to exploit the information provided by the XML schema like the disjunction nodes and the maxoccurence of an edge.

XKeyword stores the XML data in a relational database [7, 19, 10, 16, 8, 18, 5], to allow the addition of structured querying capabilities in the future and leverage the indexing capabilities of the DBMS's. Some of these works [10, 16, 8] did not assume knowledge of an XML schema. In particular, the Agora project employed a fixed relational schema, which stores a tuple per XML element. This approach is flexible but it is much less competitive than the other approaches, because of the performance problems caused by the large number of joins in SQL queries. XKeyword is different because it exploits the schema information to store the relationships between the target object id's of the XML data. The actual data are stored in XML BLOB's which are introduced in [5].



(a) Query page (b) Presentation as a list of results



(c) Presentation using Presentation graphs

Figure 4. XKeyword demo

3 Framework and Proximity Keyword Query Semantics

We use the conventional labeled graph notation to represent XML data. The nodes of the graph correspond to XML elements and are labeled with the tags of the corresponding elements and an optional string value. Figure 1 shows an example of an XML graph. An edge of the graph denotes either containment (e.g., any "*person* → *name*" edge) or an IDREF-to-ID relationship (e.g., any "*supplier* → *person*" edge) or a cross-document XML Link [21]. We will collectively refer to IDREF-to-ID and XML Link edges as *reference edges* and to the rest as *containment edges*.

We allow the graph to have multiple roots, i.e., multiple nodes with no incoming containment edge, for two reasons: First, the administrator may choose to omit the root of an XML document from the graph, since the root often provides an artificial connection between semantically unrelated first level elements. For example, had we included a root in Figure 1 it would appear as persons and parts are closely connected (two edges way) via the root; such a connection would be artificial. A second reason for multiple roots is that we may want the graph to capture multiple XML documents, potentially linked via cross-document XML Links. We also assume that every node has a unique id, invented by the system if the corresponding element has no ID attribute. Note that the graph does not consider any notion of order among the nodes $v_, \ldots, v_n$ pointed by a parent node v. In summary:

Definition 3.1 (XML graph) An XML graph G is a labeled directed graph where every node v has a unique id $id(v)$, a label $\lambda(v)$ coming from the set of element tags T and optionally a value $val(v)$ coming from the set of values V. Edges are classified into containment and reference edges. ♢

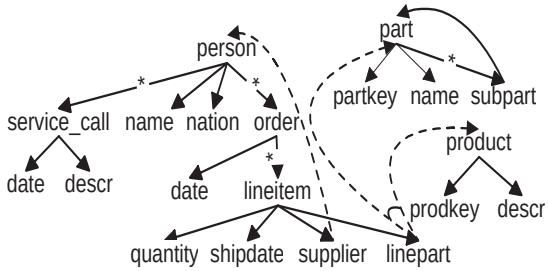


Figure 5. TPC-H based schema graph

Figure 1 illustrates an XML graph. By convention, we indicate containment edges with solid lines and reference edges with dotted lines. We omit id's from the figures and we include the values in brackets.

Schema Graphs We use *schema graphs*[2] to describe the structure of the XML graphs. Schema graphs are similar to XML Schema definitions [22] but have typed references. We have simplified the content types captured by an XML Schema and kept only the constructs that are useful for performance optimization.

The data of Figure 1 *conform* [2] to the TPC-H-like schema of Figure 5, where dotted lines denote reference edges and solid lines stand for containment edges. We denote `choice` nodes with an arc over their outgoing edges; all other nodes are of type `all`. In Figure 5, only "linepart" is

[2]defined formally in [14]

a `choice` node. Finally, we define an *uncycled directed graph* $G(V, E)$ to be a directed graph, whose equivalent undirected graph $G_u(V, E')$ has no cycles. An edge (v_1, v_2) is created in G_u if G has edges (v_1, v_2) or (v_2, v_1).

3.1 Semantics of Keyword Queries and Presentation of Results

A *keyword query* is a set of keywords $k_1, \ldots, k_m$. The result of a keyword query is the set of all possible *Minimal Total Target Object Networks (MTTON's)*. We define MTTON's after we have first defined minimal total node networks (MTNN's). A *node network* j of an XML graph G is an uncycled subgraph of G, such that for each edge $(n_1, n_2) \in j$ it is is $(n_1, n_2) \in G$. A *total node network* j of the keywords $\{k_1, \ldots, k_m\}$ is a node network, where every keyword k is contained in at least one node n of j, i.e., $\forall k \in \{k_1, \ldots, k_m\}$, $\exists n \in j : k \in keywords(n)$, where $keywords(n)$ is the set of keywords contained in the tag or the value of n. A *Minimal Total Node Network (MTNN)* j of the keywords $\{k_1, \ldots, k_m\}$ is a total node network where no node can be removed and j still be a total node network. The *score* of a MTNN j is its size in number of edges. For example the following MTNN N_0 of the keyword query "John, VCR" has size 8.

N_0 : *name*[*John*] ← *person* ← *supplier* ← *lineitem* → *linepart* → *part* → *subpart* → *part* → *name*[*VCR*]

Notice that in the general case, the size of the MTNN's of a keyword query is only data bound. Hence the user specifies the maximum size Z of an MTNN that is of interest to him/her.

To ensure that the result of a keyword query is semantically meaningful for the user we introduce the notion of *target objects*. For every node n in the XML graph we define (using the schema, as we will see later) a segment of the XML graph, called *target object* of the node n (or simply called target object when the node n is obvious from the context.) Intuitively, a target object of a node n is a piece of XML data that is large enough to be meaningful and able to semantically identify the node n while, at the same time, is as small as possible. For example, consider the MTNN N_0 above. The user would like to know which is the part number of the VCR, which is the part p of which the VCR is a subpart, which line item includes p, and what is the last name of John.[3]

The target objects provide us such information. It makes sense to output the "partkey" of the VCR part as well as the name and "partkey" of the TV. On the other hand it would not make sense to output all the subparts of the TV or the orders of the person. They could be too many and of no interest in semantically identifying the node. Hence, we define the person element with the name and nation subelements to be a target object, and the part with the "partkey" and name to be another target object.

Given a MTNN j with nodes $v_1, \ldots, v_n$ there is a corresponding MTTON t,[4] which is a tree whose nodes is a minimal set of target objects $\{t_1, \ldots, t_m\}$ such that for every node $n_k \in j$ there is a $t_l \in t$ such that *target*$(n_k) = t_l$. There is an edge from a target object t_i to a target object t_j if there is an edge (or as path of *dummy nodes* as defined below) from a node that belongs to t_i to a node that belongs to t_j. The score of a MTTON t is the score (size) of its corresponding MTNN. The answer to a keyword query is unique.

Specification of Target Objects The target objects are defined from an administrator using the *Target Schema Segment (TSS)* graph described next. A *TSS* graph is an uncycled graph whose nodes are called target schema segments. The TSS graph is derived from a partial mapping of the nodes of the schema graph G. A node t_S is created in G_{TSS} for each set $S = \{s_1, \ldots, s_w\}$ of nodes of G that are mapped to t_S. Some nodes in G, which are called *dummy schema nodes*, are not mapped to any node in G_{TSS}, because they do not carry any information. For example $supplier$, $subpart$ and $linepart$ are dummy schema nodes. An edge $(t_S, t_{S'})$ is created in G_{TSS} if the schema graph has nodes $s \in S$ and $s' \in S'$, that are connected directly through an edge (s, s') or indirectly through a path of dummy schema nodes. Typically we assign to a node t_S of the TSS graph a name that is the label of the "most representative" schema graph node $s \in S$. For example, the TSS node corresponding to $\{$*person*, *name*, *nation*$\}$ is named *person* (see Figure 6).

Figure 14 illustrates the TSS graph behind our DBLP demo. Notice the semantic explanations, with the obvious meanings, that annotate the edges. Each edge is annotated with two semantic explanations: the first explains the connection in the direction of the edge and the second in the reverse direction. Similarly, the semantic explanations of the TPC-H TSS graph are shown in Figure 6.

Given the TSS graph, it is straightforward to define a *target decomposition* of the XML graph into target objects, connected to each other. For example a target object decomposition of the schema of Figure 5 and the corresponding TSS graph are shown in Figure 6. The MTTON of the MTNN N_0 presented above is highlighted in Figure 1.

3.2 Presentation Graph

In its simplest result presentation method (Figure 4 (b)) XKeyword spawns multiple threads, evaluating various plans for producing MTTON's, and outputs MTTONs as they come. The smaller MTTON's, which are intuitively more important to the user, are usually output first, since they require smaller execution times. The threads fill a queue with MTTONs, which are output to the user page by page as in web search engine interfaces.

The naive presentation method described above (and currently used by the DBLP demo) provides fast response times, but may flood the user with results, many of which are trivial. In particular, as we explained in the introduction, a redundancy similar to the one observed in multivalued dependencies emerges often. Displaying to the user results involving multivalued dependencies is overwhelming and counter-intuitive. XKeyword faces the problem by providing an interactive interface (and corresponding API) that allows navigation and hides the trivial results, since it does not display any duplicate information as we show below.

XKeyword's interactive interface presents the results grouped by the candidate networks (see Section 4) they con-

[3] Due to space limitations we do not include a last name field in the figures.

[4] The definition does not guarantee the uniqueness of the MTTON t. The nodes of j may be split in minimal sets of target objects in multiple ways. However, this is of limited practical importance since in practice it is unlikely that target objects overlap with each other in ways that enable a network to be split in multiple ways in target objects.

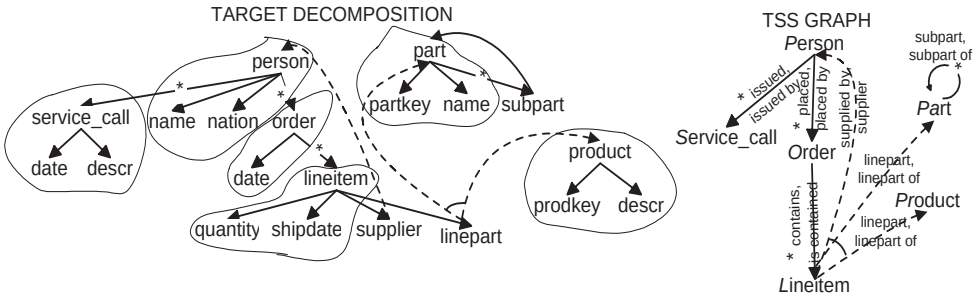


Figure 6. Target decomposition of a schema graph

form to. Intuitively, MTTON's that belong to the same candidate network have the same types of target objects and the same type of connections between them. XKeyword groups the results for each candidate network to summarize the different connection types (schemata) between the keywords and to simplify the visualization of the result.

XKeyword compacts the results' representation and offers a drill-down navigational interface to the user. In particular, a *presentation graph* $PG(C)$ (Figure 3) is created for each candidate network C. The presentation graph contains all nodes that participate in some MTTON of C. A sequence of subgraphs $PG_0(C), \ldots, PG_n(C)$ are *active* and are displayed at each point, as a result of the user's actions. The initial subgraph, $PG_0(C)$, is a single, arbitrarily chosen MTTON m of C, as shown in Figure 3 (a).

An expansion $PG_{i+1}(C)$ of $PG_i(C)$ on a node n of type N is defined as follows. All distinct nodes n', of type N, of every MTTON m' of C are displayed and marked as *expanded* (Figure 3 (b)). Note that we have to consider the statement "of type N" in a restricted sense: A candidate network may involve the same schema type in more than one roles (as is the case with tuple variable aliases in SQL.) For example, in Figure 3 "part" objects on the right side are VCRs while the "part" objects to their left are the "part" that contain the VCR parts. We consider those two classes of "part" objects to be two different types as far as presentation graphs are concerned. In addition a minimal number of nodes of other types are displayed, so that the expanded nodes appear as part of MTTON's. More formally, given a presentation graph instance $PG_i(C)$, its expansion $PG_{i+1}(C)$ on a node n of type N has the following properties: (a) $PG_i(C)$ is a subgraph of $PG_{i+1}(C)$, (b) for each MTTON $m' \in C$, where the node $n' \in m'$ is of type N, n' is included in $PG_{i+1}(C)$, (c) for each node $v \in PG_{i+1}(C)$ there is a MTTON z contained in $PG_{i+1}(C)$, such that $v \in z$, and (d) there is no instance $PG'_{i+1}(C)$ satisfying the above properties and the set of nodes of $PG'_{i+1}(C)$ is subset of the nodes of $PG_{i+1}(C)$.

In the implementation of XKeyword, an expansion on a node n occurs when the user clicks on n. Notice also that if the expanded nodes are too many to fit in the screen then only the first 10 are displayed.

On the other hand, a contraction $PG_{i+1}(C)$ of $PG_i(C)$ on an expanded node n of type N is defined as follows. All nodes of type N, except for n, are hidden (Figure 3 (c)). In addition a minimum number of nodes of types other than N are hidden, while satisfying the restriction that for each node in $PG_{i+1}(C)$ there is a containing MTTON in $PG_{i+1}(C)$ (see condition (c) below). More formally, given a presentation graph instance $PG_i(C)$, its contraction $PG_{i+1}(C)$ on an expanded node n of type N has the following properties: (a) $PG_{i+1}(C)$ is a subgraph of $PG_i(C)$, (b) n is the only node in $PG_{i+1}(C)$ of type N, (c) for each node $v \in PG_{i+1}(C)$ there is a MTTON z contained in $PG_{i+1}(C)$, such that $v \in z$, and (d) there is no instance $PG'_{i+1}(C)$ satisfying the above properties while $PG'_{i+1}(C)$ has more nodes than $PG_{i+1}(C)$. In the implementation of XKeyword, similar to the expansion, a contraction on an expanded node n occurs when the user clicks on n.

The presentation graphs model allows the user to navigate into the results without being overwhelmed by a huge number of similar MTTON's. Furthermore, if he/she is looking for a particular result it is easy to discover it by focusing on one node at a time.

The presentation of the results of a keyword query by the interactive presentation graphs evokes the following requirements for the execution unit: First the top MTTON of each candidate network, which is the initial presentation graph, must be computed very quickly to provide a quick initial response time to the user. Second the expansion of the presentation graph must be performed on demand. This cannot be done simply by moving the cursor of some query we submit to the underlying database. Instead, when a user clicks on a node, a new minimal set of focused queries is sent to the database. These requirements and corresponding solutions are addressed in Section 6.

4 Architecture

The architecture of XKeyword (Figure 7) consists of a load stage, where the data are loaded to the system and all precomputations are performed, and a query processing stage that answers keyword queries.

In the *load stage*, the $decomposer$ inputs the schema graph, the TSS graph and the XML graph and creates the following structures:

1. A *master index*, which stores for each keyword k a list of triplets of the form $\langle TO\_id, node\_id, schema\_node \rangle$ where TO_id is the id of the target object that contains the node of type $schema\_node$ with id $node\_id$, which contains k. The $node\_id$ [5] and $schema\_node$ are needed when calculating the score of a candidate network (Definition 4.1), since as we describe below, the generated relations only store target object id's.
2. A set of statistics specifying: (a) the number $s(S)$ of nodes of type S in the XML graph and (b) the average

[5] $node\_id$ is needed to distinguish two nodes of the same type and of the same target object.

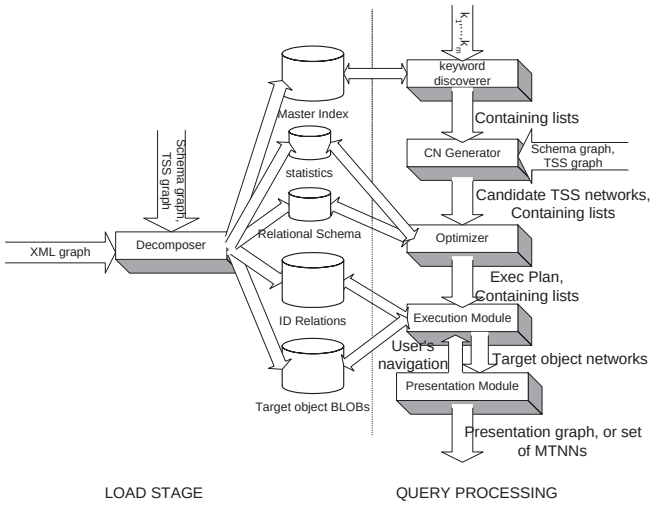


Figure 7. Architecture

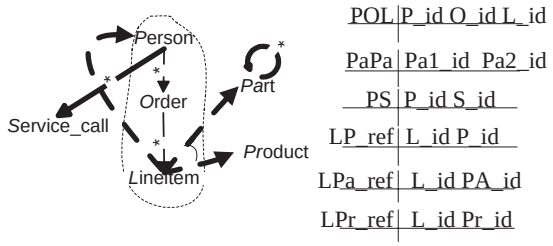


Figure 8. TSS graph Decomposition

number $c(S \rightarrow S')$ of children of type S' for a random node of type S.

3. *BLOBs of target objects*, which given an object id instantly return the whole target object.
4. A decomposition of the TSS graph into *fragments*, which correspond to *connection relations* that allow efficient retrieval of MTTON's.

Figure 8 shows a valid decomposition of the TSS graph of Figure 6, where the thick arrows and the closed dotted curves denote single edge and multiple edge fragments respectively. We map each fragment into a connection relation. For example, $P \rightarrow O \rightarrow L$ (in short *POL*, since the arrows are unambiguously implied by the TSS graph) is the connection relation that corresponds to the fragment in the dotted line. It stores the connections among the *Person*, *Order* and *Lineitem* TSS's. *LPref* is the connection relation that corresponds to the fragment (indicated by the thick dotted line) containing the reference edge between *Lineitem* and *Person*.

Query processing consists of five stages. The *keyword discoverer* inputs the set of keywords and outputs for each keyword k the *containing list* $L(k)$ of $\langle TO\_id, node\_id, schema\_node \rangle$ where the node identified by $node\_id$ contains k.

The *CN Generator* takes from the containing lists the information about which schema nodes contain the keyword and works on the schema graph to calculate all possible *candidate networks (CN's)*) (Definition 4.1). The CN Generator works on the schema graph and not on the TSS graph because (a) important schema information like the choice nodes may be lost when we create the TSS graph and (b) the score of the MTTON's is measured in terms of schema graph edges.

A *schema node network* is an uncycled directed graph of schema nodes, where for each edge (S_1, S_2) of adjacent schema nodes S_1, S_2 there is an edge (S_1, S_2) in the schema graph. The same edge of the schema graph may appear more than once in a schema node network. Intuitively, this corresponds to the fact that target objects of the same type may be playing different roles in the MTTON's. A schema node S is free (S) if its corresponding extension has nodes that contain keywords. Otherwise it is non-free. The non-free schema node S^K is the set of nodes *of type* S that contain all keywords in K.

A node network j $belongs$ to a schema node network N ($j \in N$) if there is a tree isomorphism mapping h from the nodes of j to the schema nodes of N, such that for each node $n \in j$, $n \in h(n)$.

Definition 4.1 (**Candidate Network**) Given a keyword query $k_1, \ldots, k_m$ and a schema graph, a schema node network C is a *candidate network (CN)*, if there is an instance of the XML graph that conforms to the schema graph and has a MTNN $m \in C$. ♢

The CN generator algorithm is based on the algorithm described in DISCOVER [13]. It has been extended to address the unique features of XML (choice nodes, distinction of containment and reference edges) and also incorporates performance improvements over [13]. The algorithm is presented in [14]. The CN Generator algorithm is complete (the proof is an extension of the proof of [13]), that is, all MTNN's of size up to Z belong to an output CN. Furthermore, the algorithm is non-redundant, that is, for each output candidate network C there is an instance of the database that contains a MTNN $j \in C$ and there is no other candidate network C' such that $j \in C'$.

Recall that the connection relations store only target object id's. Hence we reduce the candidate networks to *TSS networks*, which are uncycled directed graphs of TSS's, where for each edge (T_1, T_2) between TSS's T_1, T_2 there is an edge (T_1, T_2) in the TSS graph. The unique TSS network that corresponds to a candidate network is called *candidate TSS network (CTSSN)*.

The candidate TSS networks corresponding to the candidate networks of size up to $Z = 8$ are the following, where $T^{k,S}$ denotes a TSS T that contains keyword k in its schema node S:

CTSSN1: $Part^{TV,name} \rightarrow Part^{VCR,name}$
CTSSN2: $Part^{TV,name} \rightarrow Part \rightarrow Part^{VCR,name}$
CTSSN3: $Part^{TV,name} \rightarrow Part \rightarrow Part \rightarrow Part^{VCR,name}$
CTSSN4: $Part^{TV,name} \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Part^{VCR,name}$
CTSSN5: $Part^{TV,name} \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Product^{VCR,descr}$

The candidate TSS networks are output by the CN Generator. The *Optimizer* is an adaptation of the optimizer of [13] and is presented in [14] due to lack of space. It uses the schema information on the connection relations and the available statistics to generate the best *Execution Plan* that evaluates the set of candidate TSS networks.

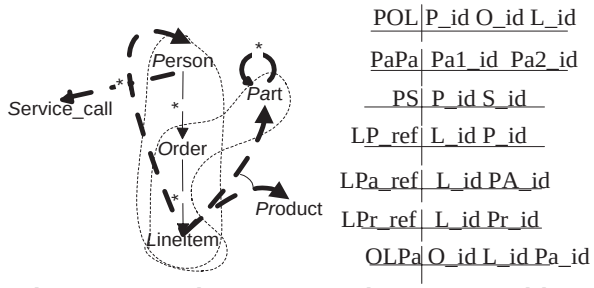


Figure 9. Another TSS Graph Decomposition

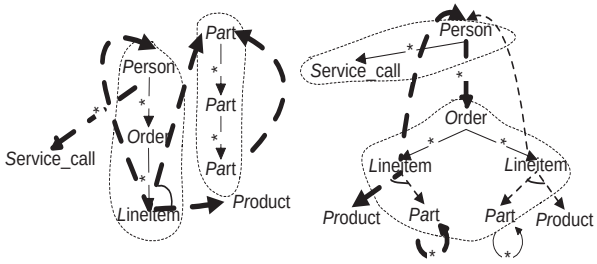


Figure 10. Unfolded TSS Graph Decompositions

The *Execution Module* inputs the execution plan from the Optimizer. If the presentation graph method is used to present the results, the Execution Module interacts with the *Presentation Module* to direct the execution according to the user's navigation on the presentation graphs. In the case of the full list of results presentation method, a stream of results is output. The details of the Execution Module are described in Section 6.

Finally the Presentation module displays the results as described in Section 3.2.

5 XML Decompositions

The decomposition of the TSS graph into fragments determines how the connections of the XML graph are stored in the database, and consequently the generated execution plan for the candidate TSS networks. We have found that the selected decomposition can dramatically change the performance of XKeyword, especially for top-K queries.

EXAMPLE 5.1 Consider the keyword query "TV, VCR" and $CTSSN4$: $Part^{TV,name} \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Part^{VCR,name}$ from Section 4. $CTSSN4$ requires three joins given the decomposition of Figure 8. Consider the TSS graph decomposition of Figure 9, which includes an $OLPa$ fragment. With this decomposition, $CTSSN4$ can be evaluated with a single join $OLP^{TV,part.name} \bowtie OLP^{VCR,part.name}$. □

Often we need to build $unfolded$ fragments that contain the same TSS more than once, to store the same edge of the TSS graph more than once, as shown in the example below.

EXAMPLE 5.2 Consider the network $CTSSN2$: $Part^{TV,name} \rightarrow Part \rightarrow Part^{VCR,name}$ of Section 4. This network connects three $Part$ nodes by following the $Part \rightarrow Part$ edge twice. Under any non-unfolded decomposition this network cannot be executed without a join. However, the first unfolded TSS graph of Figure 10, which "unrolls" the $PartPart$ cycle, allows the creation of the $Part \rightarrow Part \rightarrow Part$ fragment, which can evaluate $CTSSN2$ without a join.

Similarly, $CTSSN4$ can be evaluated without a join, if we create the $Part \leftarrow Lineitem \leftarrow Order \rightarrow Lineitem \rightarrow Part$ fragment on the second unfolded TSS graph of Figure 10, where the $Order \rightarrow Lineitem$ edge has been "split", i.e., the $Order$ TSS has two children $Lineitem$ TSS's. Notice that not all edges of the unfolded TSS graphs have to be in the decomposition. For example in the second unfolded TSS graph of Figure 10, the second $Lineitem \rightarrow Person$ edge is not in a fragment, since there is a fragment for the first $Lineitem \rightarrow Person$ edge. □

Definition 5.1 (**Walk Set, Unfolded TSS Graph**) A walk set of a TSS graph G, denoted $WS(G)$, is the set of all possible walks in G. A graph G_u is an unfolded TSS graph of the TSS graph G if $WS(G_u) = WS(G)$. ♢

Definition 5.2 (**TSS Graph Decomposition**) A decomposition of a TSS graph $G = \langle N, E\rangle$ is a set of fragments $F_1, \ldots, F_n$, where for each fragment $F\langle N, E\rangle$ there is an unfolded TSS graph $G_u = \langle N_u, E_u\rangle$ of G, such that F is a subgraph of G_u. Every edge of G has to be present in at least one fragment. ♢

Lemma 5.1 *Any candidate TSS network can be evaluated given a TSS graph decomposition with the properties of Definition 5.2.*

The size of a fragment is the number of edges of the TSS graph that it includes. Note that a TSS graph decomposition is not necessarily a partition of the TSS graph – a TSS may be included in multiple fragments (Figure 9).

Each fragment $F = \langle N, E\rangle$ corresponds to a *connection relation* R, where each attribute corresponds to a TSS and is of type ID[6]. A tuple is added to R for each subgraph of type F in the *target object graph*, which is the representation of the XML graph in terms of target objects, that is, each node of the target object graph is a target object. Connection relations are a generalization of *path indexes* [9].

5.1 Decomposition Tradeoffs

There is a tradeoff between the number of fragments that we build and the performance of the keyword queries, as we shown in Section 7. Assume that we consider solutions to the keyword queries which contain up to $M + 1$ target objects. That is, the maximum size of a candidate TSS network is M. The one extreme is to create the *minimal decomposition*, where a fragment is built for each edge of the TSS graph. Then, each candidate TSS network C requires $S - 1$ joins to be evaluated, where S is the size of C. We have found that the minimal is the most efficient decomposition for the on-demand expansion of a presentation graph, because the execution algorithm first tries to connect the new target objects to the adjacent nodes in the presentation graph, and gradually tries further nodes (Figure 13).

The other extreme is the *maximal decomposition*, where a fragment F is built for every possible candidate TSS network C. F is created by replacing the non-free TSS's of C with free TSS's. Then C is evaluated with zero joins. Clearly, the maximal decomposition is not feasible in practice due to the huge amount of space required.

Notice that M can be calculated by the maximum size Z of the MTNN's of the keyword query. In particular, the size S of a candidate TSS network C is bound by the size S' of

[6] In RDBMS's we use the "integer" type to represent the "ID" datatype.

the corresponding candidate network C' with the *size association function* f, which depends on the schema graph, the number of keywords and the TSS graph. It is $S \leq f(S')$. Hence

$$M = f(Z) \quad (1)$$

For the schema graph of Figure 5, two keywords and the TSS graph of Figure 6, it is $f(S') = 2 \cdot S' + 2$.

The clustering and indexing of the connection relations are critical because they determine the performance of the joins. In the maximal decomposition, a multi-attribute index is created for every valid (i.e., the keywords can be on these attributes) combination of attributes of every connection relation. In all non-maximal decompositions, we found (Section 7) that the performance is dramatically improved when a connection relation R is clustered on the direction that R is used. For example, consider the execution plan of Section 4. If the evaluation of $CTSSN3 \leftarrow PaPa^{(TV,part1.name)} \bowtie_{Pa2\_id=Pa1\_id} PaPa \bowtie_{Pa2\_id=Pa1\_id} PaPa^{(VCR,part2.name)}$ starts from the left end, then all three $PaPa$ connection relations should be clustered from left to right. If creating all clusterings for each fragment is too expensive with respect to space, then single attribute indices are created on every attribute of the connection relations, since we found that multi-attribute indices are not used by the DBMS optimizer to evaluate join sequences.

The number of joins to evaluate the query q corresponding to a candidate TSS network is critical, because of the nature of q, which always starts from "small" connection relations. Also, the connection relations only store ID's and have every single attribute index, which makes the joins index lookups. The significance of the number of joins was verified experimentally (Section 7). Hence, we specify for each decomposition an upper bound B to the number of joins to evaluate any candidate TSS network of size up to M. For example $B = 0$ and $B = M - 1$ for the maximal and minimal decompositions respectively.

Given B, we generally prefer to build fragments of small sizes to limit the space of storing them. Theorem 5.1 proves that we can bound the size of the fragments of the decomposition.

Theorem 5.1 *There is always a decomposition D, whose fragments' maximum size is $L = \lceil \frac{M}{B+1} \rceil$ and any candidate TSS network of size up to M is evaluated with at most B joins.*

Proof: See [14]. $\diamondsuit$

Depending on the TSS graph, we may need to build all possible fragments of size L to satisfy the constraint B on the number of joins. Theorem 5.2 shows such a class of TSS graphs.

Theorem 5.2 *If all edges of the TSS graph are star ("*") edges and $\exists L \in \mathbf{N}$, such that $M = L \cdot (B + 1)$, then the decomposition D must contain all fragments of size L to satisfy the constraint B on the number of joins.*

Proof: See [14]. $\diamondsuit$

Often it is not efficient to build all fragments of size L, because a fragment may take up too much space despite its small size (in number of edges). This happens when the corresponding connection relation of a fragment has a non-trivial multivalued dependency (MVD), as the $PaLOLPa$ fragment in Figure 10, which has the MVD $O\_id \twoheadrightarrow L1\_id, Pa1\_id$. We say that a fragment has an MVD when its corresponding connection relation has an MVD.

Theorem 5.3 *A fragment F has a non-trivial MVD iff F contains a path $p = (e_1, \ldots, e_n)$ and $\exists e_i \in \{e_1, \ldots, e_n\}, \exists e_j \in \{e_1, \ldots, e_n\}, i < j$, and*

- $e_i \in \{\overset{*}{\leftarrow}, \xrightarrow{ref}, \underset{ref}{\overset{*}{\rightarrow}}, \underset{ref}{\overset{*}{\leftarrow}}\}$ *and*
- $e_j \in \{\overset{*}{\rightarrow}, \xleftarrow{ref}, \underset{ref}{\overset{*}{\rightarrow}}, \underset{ref}{\overset{*}{\leftarrow}}\}$ *and*
- $\nexists l, i < l < j - 1, e_l \in \{\rightarrow\} \wedge e_{l+1} \in \{\leftarrow\}$

Proof: See [14]. $\diamondsuit$

We classify TSS graph fragments and decompositions based on the storage redundancy in the corresponding connection relations. Connection relations that correspond to a single edge in the TSS graph, by definition are always in 4NF. Some wider connection relations, for example the $OLPa$ relation of Figure 9 can be in 4NF, however most of them will not be in 4NF. Non-MVD, no-4NF connection relations, are called *inlined* connection relations. A fragment is 4NF, inlined, or MVD, if the resulting connection relation is 4NF, inlined, or MVD respectively.

There are two classes of fragments that should never be built because no candidate TSS network can efficiently use them. We call such fragments $useless$:

1. If a fragment F contains a choice TSS T and more than one children of T, then F is useless, since the children of T can never be connected through T. For example, the fragment $PaLPr$ is useless since $Lineitem$ is a choice TSS.
2. A fragment that contains the construct $T_1 \xrightarrow{l_1} T \xleftarrow{l_2} T_2$ is useless, if $l_1 \neq ref$ and $l_2 \neq ref$, because T_1 and T_2 are never connected through T. For example, the fragment L_1PrL_2 is useless since two $Lineitem$ target objects cannot connect through a $Part$ target object.

We ignore useless fragments in the decomposition algorithm presented below.

Decomposition Algorithm. XKeyword uses two different decompositions. First, an *inlined, non-MVD* decomposition generated by the algorithm of Figure 12 is built, where B is the maximum number of joins and M is the maximum candidate TSS network size. This decomposition is used to efficiently generate the top-K results (MTTON's) in the web search engine-like presentation, and the top-1 MTTON of each CN C which corresponds to the initial instance of the presentation graph of C. Second, the minimal decomposition is built, which is used along with the *inlined, non-MVD* decomposition in the on-demand expansion of the presentation graphs.

The algorithm in Figure 12:

- satisfies the B constraint on the number of joins
- avoids building MVD fragments if possible
- builds non-MVD fragments of size larger than $L = \lceil \frac{M}{B+1} \rceil$ if they can eliminate MVD fragments of size L

We say that a candidate TSS network C is $covered$ by a decomposition D when C can be evaluated with at most B joins.

Given $M = 4$ and $B = 1$, Figure 11 shows how the candidate TSS network $S \leftarrow P \rightarrow O \rightarrow L \rightarrow Pr$ is covered if we build the non-MVD fragment $POLPr$ of size $L + 1$ instead of the MVD fragment SPO of size L.

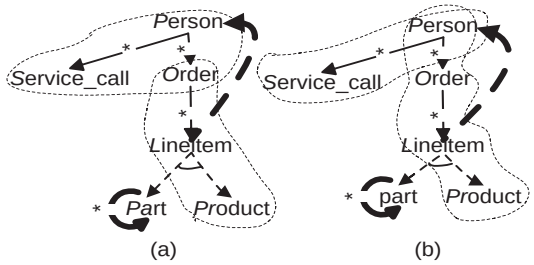


Figure 11. Replacing an MVD with a non-MVD fragment

Decomposition Algorithm(B,M){
Add to the decomposition D the non-MVD fragments of size $\leq L$;
Create a list Q of all candidate TSS networks
of size up to M not covered by D;
Add all possible non-MVD fragments of size
greater than L, that help in covering at
least one candidate TSS network $C \in Q$ and remove C from Q;
Add the minimum number of MVD fragments of size up to L
to cover all candidate TSS networks in Q;
}

Figure 12. Decomposition Algorithm

6 Execution

The execution module of XKeyword aims at providing fast response time to keyword queries. Depending on the presentation method selected (see Section 3.1), we follow a different execution approach.

Web search engine-like presentation. In the case of the web search engine-like presentation of the MTTON's (Figure 4 (b)), we use the inlined, non-mvd decomposition (Figure 12) to speedup the execution of the top-K keyword query. If the CN's[7] were evaluated sequentially, and the first one did not produce any results, then the time to get the first result would be too long. We solve this problem be using a thread pool. A thread is assigned to each CN starting from the smaller ones, which need less execution time and also produce higher ranked results. A thread is returned to the pool when either the corresponding CN's evaluation completed, or a total of K results have been generated by all threads, in which case the execution ends.

The evaluation of a single CN C of the keyword query $k_1, \ldots, k_m$ is challenging for two reasons. First, since we look for K results, sending a SQL statement for C is inefficient, because the DBMS's do not currently efficiently support top-K queries. XKeyword uses nested loops join, where the nesting of the loops is determinded by a depth first traversal of C that first finds a connection between k_1 and k_2, then to k_3, etc. The execution is terminated after K results are produced. For example, consider $CTSSN2 : part^{TV,name} \rightarrow part \rightarrow part^{VCR,name}$ of Section 4. The outermost loop will iterate over the TSS $part^{VCR,name}$[8], the second loop over $part$ and the innermost over $part^{TV,name}$.

[7]For simplicity, in this section we use the term CN for both a CN and its corresponding candidate TSS network.

[8]In the next paragraph we explain why VCR was selected as k_1.

Expansion Algorithm(PG(C),n){
$PG(C)$: current instance of presentation graph
n: node to be expanded. n is of type N
Let S be the set of target objects of type N;
for each node u in S do
$l := 1$;
while u not connected to all keywords and $l \leq size(C)$ do
Check if u is connected to all keywords through $PG(C)$
with l extra edges;
l++;
If no connection was found ignore u;
else add u with its connection edges to $PG(C)$;
}

Figure 13. On-demand expansion algorithm

The second challenge is that the $naive$ nested loops join algorithm has a serious inefficiency, because it may send the same queries multiple times. In the above example, consider the case where two target objects t_1, t_2 in $part^{VCR,name}$ connect to the same target object in the $part$ TSS. Then, when evaluating $CTSSN2$ for t_2, the innermost loop (over $part^{TV,name}$) should not be executed since it will produce the same results as before. Notice that this optimization would not be possible if we had put $part^{TV,name}$ as the outermost loop, because $part \rightarrow part$ is a containment and not a reference edge, so no two target objects in $part^{TV,name}$ could connect to the same target objects in the $part$ TSS. The speedup of the $optimized$ execution algorithm over the naive one is experimentally evaluated in Section 7.

In the optimized execution algorithm, there is a tradeoff between storing the past results to avoid repeating a query and keeping no past results but sending more queries. XKeyword uses a fixed size cache for each keyword query to store past results and if the cache gets full, the queries are re-send to the DBMS.

Presentation Graphs. In the case of the on-demand execution based on the presentation graphs' navigation we need to modify the optimized algorithm, because we do not need the complete MTTONs, but only to find the set of expanded nodes that the user requested and their minimal connections to the presentation graph. In the above example, if the user clicks on the $part^{TV,name}$ TSS, then for each expanded node n in $part^{TV,name}$, we need to find a single connection to the $part$ TSS and we ignore additional connections. In particular, we first check if n is connected to a node of $part$ already in the presentation graph $PG(CTSSN2)$, because we need to expand the $PG(CTSSN2)$ in a minimal way. If such a connection is not possible, we search for a connection to a fresh node of the $part$ TSS. The on-demand expansion algorithm is shown in Figure 13.

Initially, the XKeyword decomposition (Figure 12) is used to efficiently retrieve the top result of each CN. Then we use a combination of the minimal and the *inlined, non-MVD* decomposition to find the minimal connection of the expanded nodes to the presentation graph, as we explain in Section 7.

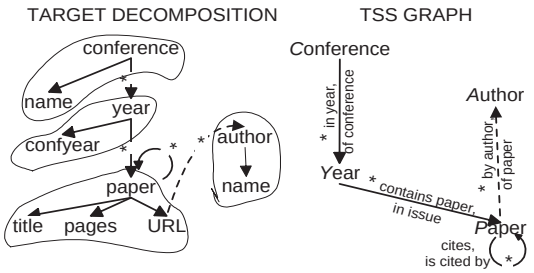


Figure 14. Target decomposition of DBLP

7 Experiments

To evaluate the performance of XKeyword we performed a set of experiments. First, we measure the performance of the keyword queries for various decompositions of the XML schema, for top-K and full results. Then we evaluate the performance of the optimized execution algorithm for the search engine-like presentation method described in Section 6. Finally the performance of the on-demand expansion algorithm is evaluated.

We use the DBLP XML database with the schema shown in Figure 14. The citations of many papers are not contained in the DBLP database, so we randomly added a set of citations to each such paper, such that the average number of citations of each paper is 20. We use Oracle 9i, running on a Xeon 2.2GHz PC with 1GB of RAM. XKeyword has been implemented in Java and connects to the underlying DBMS through JDBC. The master index is implemented using the full-text Oracle 9i interMedia Text extension. Clustering is performed using index-organized tables.

Decompositions. We assume that the maximum candidate networks' size is $Z = 8$ and focus on the case of two keywords. Notice that we select a big Z value to show the importance of the selected decomposition. The absolute times are an order of magnitude smaller when we reduce Z by one. For the TSS graph of Figure 14, the maximum size of the CTSSN's is $M = f(8) = 8 - 2 = 6$. We require that the maximum number of joins is $B = 2$, hence from Theorem 5.1 it is $L = 2$. We compare five different decompositions:

1. The $XKeyword$ decomposition created by the algorithm of Figure 12.
2. The $Complete$ decomposition, which consists of all fragments of size L.
3. The $MinClust$ decomposition, which is the minimal decomposition with all possible clusterings for each fragment.
4. The $MinNClustIndx$ decomposition, which is the minimal decomposition with single attribute indices on every attribute of the ID relations.
5. The $MinNClustNIndx$ decomposition, which is the minimal decomposition with no indices or clustering.

We compare the average performance of these decompositions to output the top-K results for each candidate network. The results are shown in Figure 15 (a). Notice that the $Complete$ decomposition is slower than $MinClust$ although it requires a smaller number of joins, because of the huge size of the fragments that correspond to relations with multi-valued dependencies and the more efficient caching performed in the $MinClust$ decomposition. Also notice that the non-clustered decompositions (the results for $MinNClustNIndx$ are not shown, because they are worse by an order of magnitude) perform poorly for the top-K results.

Figure 15 (b) shows the average execution times to output all the results for each candidate network. Notice that the $MinNClustNIndx$ is the fastest, since the full table scan and the hash join is the fastest way to perform a join when the size of the relations is small relatively to the main memory and the disk transfer rate of the system, which is the case here, since all relations of the minimal decomposition have just two id (integer) attributes.

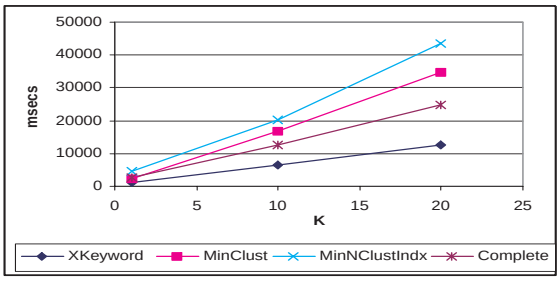
Execution Algorithm. We evaluate the optimized execution algorithm, where partial results are cached and reused (and hence production of "trivial" results, due to multi-valued dependencies, is reduced) for the search engine-like (non-interactive) presentation method. We measure the speedup of the optimized algorithm compared to the naive, non-caching algorithm for various candidate TSS network sizes M. The results are shown in Figure 16 (a), where the number of keywords is fixed to 2. We see that the speedup increases with M because the number of trivial results increases with M. Also notice that the speedup is smaller than 1 for $M = 2$, because of the negligible caching opportunities and the overhead imposed by the caching algorithm.

Finally, we measure the performance of the on-demand expansion algorithm of Figure 13. We use keyword queries that involve the names of two authors and we focus on the presentation of results coming from the candidate network $Author^{k_1} \leftarrow Paper \rightarrow Author^{k_2}$.Figure 16 (b) shows the average time to expand a $Paper$ node using three different decompositions: (i) the inlined, non-mvd decomposition produced by the algorithm of Figure 12, (ii) the minimal decomposition, and (iii) the combination of the two decompositions, i.e., the union of their fragments. More internal $Paper$ nodes are added for bigger sizes. The combining decomposition is faster when the size of the candidate TSS networks is greater than 2. It is slightly slower than the minimal for size 2, due to the caching of the minimal connection relations by the DBMS. The inlined is slower, because the algorithm initially looks for a connection to the adjacent nodes of the to be expanded node, where the minimal fragments are more suitable.

8 Conclusions and Future Work

XKeyword is a system that offers keyword proximity search on XML databases that conform to an XML schema. The XML elements are grouped into target objects, whose connections are stored in connection relations. Redundant connection relations are used to improve the performance of top-K keyword queries. XKeyword presents the results as interactive presentation graphs, which summarize the results per candidate network. The execution of the queries is optimized to offer fast response times.

In the future, we plan to look into different semantics for keyword queries on structured and semi-structured databases, going beyond the distance between keywords. We also work on integrating the master index tighter into the execution engine of XKeyword and on improving the response time of the system.

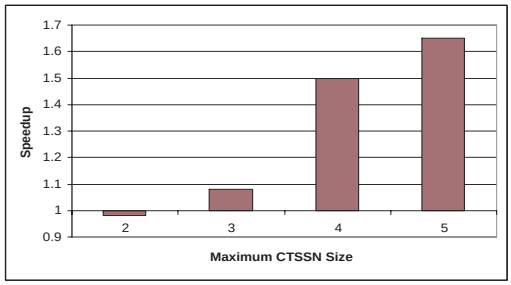


(a) Top-K results

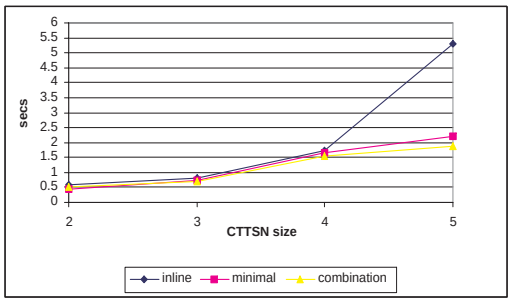
Decomposition	secs
XKeyword	170.8
Complete	302.9
MinClust	225.2
MinNClustIndx	347.4
MinNClustNIndx	137.3

(b) All results

Figure 15. Execution times



(a) Speedup by caching partial results



(b) Expansion of presentation graph

Figure 16. Execution times

9 Acknowledgements

We thank Patrick Lightbody for implementing the front end of the XKeyword demo. We also thank Tianqiu Tempo Wang for implementing the module that displays the presentation graphs and helping with the experiments.

References

[1] http://www.xyzfind.com.

[2] S. Abiteboul, D. Suciu, and P. Buneman. Data on the Web : From Relations to Semistructured Data and Xml. *Morgan Kaufmann Series in Data Management Systems*, 2000.

[3] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System For Keyword-Based Search Over Relational Databases. *ICDE*, 2002.

[4] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. *International Conference on Extending Database Technology (EDBT)*, 2002.

[5] A. Balmin and Y. Papakonstantinou. Storing and Querying XML Data Using Denormalized Relational Databases. *UCSD Technical Report*, 2001.

[6] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. *ICDE*, 2002.

[7] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of ICDE*, 2002.

[8] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. *ACM SIGMOD*, 1999.

[9] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE*, pages 14–23, 1998.

[10] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[11] D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. *WWW9 Conference*, 1999.

[12] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. *VLDB*, 1998.

[13] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. *VLDB*, 2002.

[14] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs (extended version). *UCSD Technical Report*, 2002.

[15] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. *PODS*, 2001.

[16] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with XML and relational. *VLDB*, 2000.

[17] J. Plesn'ik. A bound for the Steiner tree problem in graphs. *Math. Slovaca 31*, pages 155–163, 1981.

[18] A. Schmidt, M. L. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *WebDB (Selected Papers)*, pages 47–52, 2001.

[19] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. *VLDB*, 1999.

[20] J. D. Ullman, J. Widom, and H. Garcia-Molina. Database Systems: The Complete Book. *Prentice Hall*, 2001.

[21] W3C. XML Linking Language (XLink), 2001. W3C Recomendation available at http://www.w3.org/TR/xlink/.

[22] W3C. XML schema definition, 2001. W3C Recomendation available at http://www.w3c.org/XML/Schema.

[23] W3C. XQuery: A query language for XML, 2001. W3C Working Draft available at http://www.w3c.org/XML/Query.