Concept based querying of semistructured data

Margret Groß-Hardt

margret@uni-koblenz.de

Abstract: In the last years, semistructured data has played an increasing role within the database community. Many query languages have been developed for querying semistructured data and in particular XML data sources. XML data often is described by means of DTDs and more recently through XML schemas. This paper is about querying semistructured data by making use of the schema and the types described therein. Elements of an XML documents are considered as instances of schema concepts. *Concept based queries* provide a means to retrieve instances based on concept names and in particular offer a possibility to exploit generalization relationships between concepts. As a consequence, concept based queries free users from knowing the details about the structure in XML documents and hence ease querying of semi structured data.

1 Introduction

In the last years, semistructured data has played an increasing role within the database community. Many query languages have been developed for querying semistructured data and in particular XML data sources.

As pointed out in [Abi97], semistructured data often has irregular structure, is implicit, incomplete and as a consequence results in large schemas. As a consequence, in order to query semistructured data, queries over the schema are as important as standard queries over the data because the user is not supposed to know all the details of the schema. In the case of XML data, a schema either is given as a DTD and more recently by means of XML Schema [XSc01]. One important restriction of DTDs is, that with DTDs user defined types are not provided and in terms of conceptual abstractions [BCN92], aggregation and association relationships can be represented but generalization / specialization relationships are not supported. XML Schema adds user defined types and by means of so called *substitution groups* and *extension base* constructs the possibility to represent generalization / specialization relationships.

For instance, consider an XML data source that represents information about departments in a university, their researchers with their publications and the university library with its books. If we are interested in all publications either listed as articles or monographs of researchers or listed as books in the library, we need to know the details of the element names and the nesting level within the complete document to access this information.

This paper is about querying semistructured data by making use of the schema and the

types described therein. Elements of an XML documents are considered as instances of schema types, in this paper also called *concepts*. *Concept based queries* provide a means to retrieve instances of certain concepts. With respect, to the university example, a concept based query could be: “Show all elements belonging to the concept publication”. Concept based queries exploit typing knowledge available in the schema. This approach extends existing query languages and offer a concise mechanism to address “similar” elements belonging to the same (general) type. Using schema information, additionally, helps in detecting query errors at compile time, rather than at run time [DFF+99].

Query languages [DFF+99, KS01, ML01, XQu01] proposed for semistructured data and in particular for XML based data sources are defined on the XML instance, sometimes partly based on DTDs, too; hence they do not support abstraction concepts like generalization. These query languages make extensive use of path expressions, to navigate through nested structures.

We propose a query language that may contain schema expressions. Like in object oriented languages [CACS94, Cat96], too, our query language support modeling constructs like aggregation and generalization. Queries can be written against this schema, refer to schema concepts, e.g. the type of an element and then are translated automatically into one or more “classical” path expressions, no longer containing any schema expressions. These generated path expressions either could be offered to the user, who selects a subset of these expressions to be executed against the underlying XML data source or they are evaluated on the data, directly.

The benefit of this approach is, that users only have to have an idea about the concepts stored in the data source, but they do not have to know the details about element names and nesting structures.

1.1 Example

Figure 1 shows an extract of an XML document which describes example publication data available in a university. A university might consists of eventually multiple departments and a library. A department has members which are researchers and researchers are described by means of their name and information about their publications. A library consists of many books which has title and author information. Note, this example illustrates the different kinds of publication data represented in the university either as part of a researcher or as books in the library.

A closer look to the content of a `researcher` element in this example reveals three different subelements which describe publications of researchers. In particular, there are subelements `publication`, `monograph` and `article`. All these elements as well as `book` and `monograph` contain title and author data and some additional publication type specific information.

If someone wants to access the different kinds of publications available in the data, he needs all relevant element names. XPath, for instance, allows a query like “`//publication $union$ //book $union$ //monograph $union$ //article`”. So, even if a user

```
<university>
  <researcher>
    <name>Smith</name>
    <publications>
      <monograph>
        <title>Basics of databases</title>
        <author><name>Smith</name>
        </author>
        <isbn>7899</isbn>
        <subject>DB</subject>
      </monograph>
      <article>
        <title>Flexible queries</title>
        <author><name>Smith</name>
        </author>
        <author><name>Miller</name>
        </author>
        <proceeding>VLDB 2000
        </proceeding>
      </article>
      <publication>
        <title>XML Schema</title>
        <author>
          <name>Smith</name>
        </author>
      </publication>
    </publications>
  </researcher>
  <-- more researcher elements -->
  <library>
    <books>
      <book>
        <title>Databases</title>
        <author>
          <name>Smith</name>
        </author>
        <isbn>1234</isbn>
      </book>
      <book>
      <!-- more book elements -->
      </book>
    </books>
  <library>
</university>
```

Figure 1: Example XML data

does not have to know the details of the paths from the root to the elements themselves[1], he has to know all names of elements that has “something to do” with publications.

A possible XML schema for the given XML data source is indicated in Figure 2. Note, for every complex element, a named complex type is defined within the schema and each complex element is decribed by such a type. For instance, the `university` element is described by the type `UNIVERSITY` and e. g. `publication` is described by the type `PUBLICATION`. We will write element names in a XML data source in small letters and we will use capital letters for type names in the XML schema. In particular, the set of element names and the set of type names have to be disjoint.

There are two aspects within an XML schema, that we want to concentrate on here: These are type derivation and the possibility to build substitution groups [XSc01].

Types are partially ordered by means of so called *extension base* constructs in the XML schema. E.g., `BOOK` and `ARTICLE` are derived from `PUBLICATION` and `MONOGRAPH` from `BOOK`. Derived types consist of all elements defined in the more general type and may add additional elements. E.g. `BOOK` contains the same elements as `PUBLICATION` and adds an element `isbn`.

The second modeling construct, that is has been used in the given schema is the *substitution group*. For instance, in Figure 2, there is a substitution group used within the definition of a `researcher` element. The substitution group introduced in the beginning of the schema specifies, that the *head* of the substitution group is the element `publication` and possible *substitutes* are `book` and `article` elements. For a valid substitution group, the type of a substitute element has to be derived by the type of the head element [XSc01].

[1] “//” is an expression that stands for “all descendents of the current node”, see [XPa99].

```
<schema ...>
<element name="university"
         type="UNIVERSITY"/>
<element name="publication"
         type="PUBLICATION"/>
<element name="monograph"type="MONOGRAPH"
         substitutionGroup="publication"/>
<element name="article"type="ARTICLE"
         substitutionGroup="publication"/>
<complexType name="UNIVERSITY">
  <sequence>
    <element name="name"type="string"/>
    <element name="researcher"
             type="RESEARCHER"
             maxOccurs="unbounded"/>
    <element name="library"
             type="LIBRARY"/>
  </sequence>
</complexType>
<complexType name="LIBRARY">
  <sequence>
    <element name="books"
            type="BOOKS"/>
  </sequence>
</complexType>
<complexType name="BOOKS">
  <sequence>
    <element name="book"type="BOOK"
       maxOccurs="unbounded"/>
  <sequence>
</complexType>
<complexType name="RESEARCHER">
  <sequence>
    <element name="name"type="string"/>
    <element name="publications"
       type="PUBLICATIONS"/>
  </sequence>
</complexType>
```

```
<complexType name="PUBLICATION">
  <sequence>
    <element name="title"type="string"/>
    <element name="author"type="AUTHOR"
       maxOccurs="unbounded"/>
  </sequence>
</complexType>
<complexType name="MONOGRAPH">
  <complexContent>
    <extension base="BOOK">
      <sequence>
        <element name="subject"
                 type="string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="ARTICLE">
  <complexContent>
    <extension base="PUBLICATION">
      <sequence>
        <element name="proceeding"
                 type="string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="BOOK">
  <complexContent>
    <extension base="PUBLICATION">
      <sequence>
        <element name="isbn"
          type="string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType></schema>
```

Figure 2: XML schema excerpt related to Figure 1

1.2 Structure of this document

This paper is structured as follows: The next section introduces the data model and describes databases, schemas and so called *schema graphs* (see section 2.1). In section 3, the query language is introduced. Firstly, a strict query language is defined, which basically reflects the “usual” semantics of queries for semi structured data. On top of strict queries, concept based queries are presented. We define validity of queries w.r.t. to the schema graph and describe the processing of concept based queries by transforming them into strict queries. Section 4 gives an overview about related work and section 5 summarizes this article and indicates future work.

2 Data model

The data model distinguishes between the instance layer or database layer on the one hand side and the schema layer for the data on the other side. The database of semistructured data is described by means of the OEM data model [PGMW95, AQM^{+}97]. The concepts

and their relationship on the schema level are described by means of a schema graph. The goal of a schema representation by means of this graph is to adequately reflect subtyping relationships and substitution groups in the XML schema.

The database is represented as a labeled directed graph. Figure 3 illustrates the database

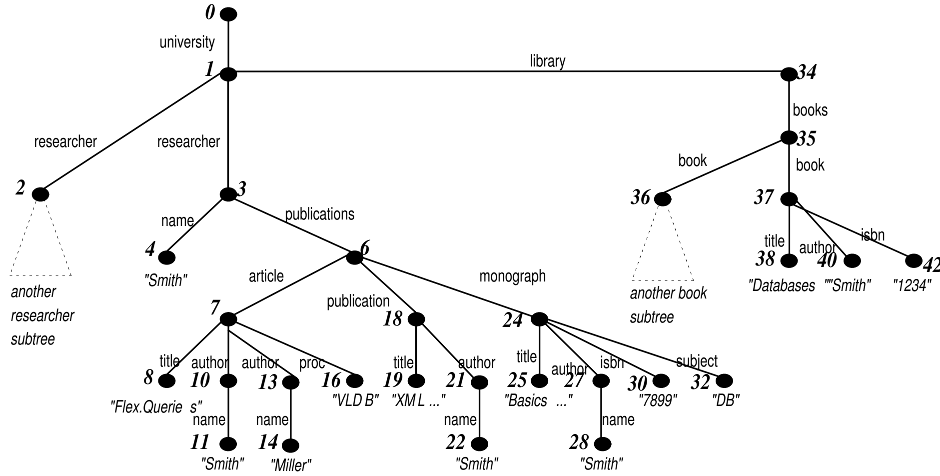


Figure 3: University example

related to the example shown in Figure 1. Each node represents an *object* and has an *oid*. There is exactly one root object representing the top level element in the XML document. Nodes without outgoing edges are called *atomic nodes*. These nodes also have a value, shown below these nodes. In this paper, we will restrict our focus to atomic nodes of type `STRING`. Nodes with outgoing edges represent *complex objects*. Every node in the database is reachable from the root.

In Figure 3, the root node refers to the top level element `university`. There is a sub tree of the `university` node for every `researcher` element and one sub tree for `library`. Analogously, there are sub trees of researcher and library nodes to describe their details, respectively.

Formally a database D is a 4-tuple $D = (O, E^D_{\mathcal{L}}, r_D, \alpha)$, where O is a finite set of objects (i.e. nodes), $E^D_{\mathcal{L}}$ is a set of labeled edges e of the form $e = (u, l, v)$ for objects u, v and label l; r_D is the root and α is a function that maps each atomic node to a value [KS01]. The set $\mathcal{L}$ of labels of $E^D_{\mathcal{L}}$ is called *element names* of D.

A path between two objects u, v is a sequence of edges. The label of a path p, *label(p)*, is a "." separated list of edge labels on p. The set of objects reachable by a path p that starts in node u and has label(p) = π is denoted *objects(u, π)*. For $u = r_D$, we also write *objects(π)* to denote all objects reachable from the root node via a path with label π.

Example 1 *Consider our example in Figure 3. Some path labels for paths starting with the root node together with the associated objects are:*

π_1 = *university.researcher* *objects*$(\pi_1) = \{2, 3\}$
π_2 = *university.researcher.publications* *objects*$(\pi_2) = \{6\}$
π_3 = *university.researcher.publications.article.title* *objects*$(\pi_3) = \{8\}$

2.1 Schemagraph

Schemagraphs graphically describe a schema. Figure 4 shows a schema graph related to the XML schema introduced in Figure 2. Types in the XML schema are represented as

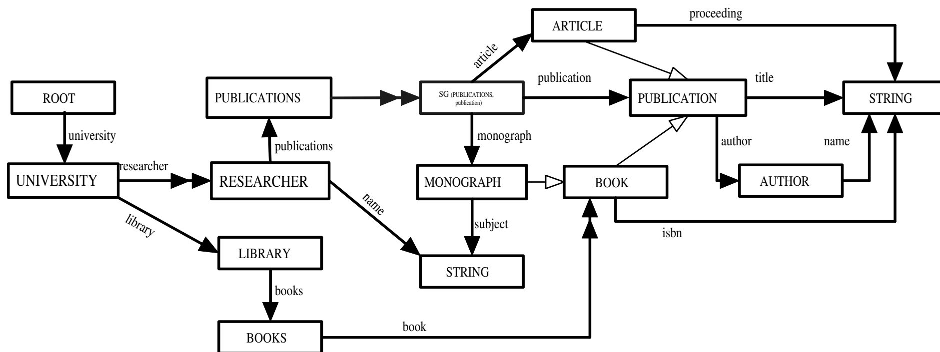


Figure 4: Schemagraph related to Figure 3

nodes in the graph; types either are atomic types, here `STRING` only, or they are complex types. There is a distinct node, representing the type of the root element, called `ROOT`. This node does not have any incoming edges and only has one outgoing edge labeled with the name of the top level element in the XML schema.

Subtype relationships between types are indicated by edges with white arrows. These edges are called *isa-edges*. A type may have no more than one supertype and every type in the schema graph not explicitly associated with its supertype stands in *isa* relationship with type `ANY` (not shown in the figure). Edges with two arrows denote a multi valued reference between types.

Substitution groups in the XML schema are represented by nodes in the schema graph. Due to the fact, that substitution groups do not have a name, the corresponding nodes in the schema graph are labeled with some artificial id not occurring otherwise as a node name. If the substitution group s is defined in type t, there is an edge without label between them. Also, for every substitute $u_1, u_2, \ldots, u_n$ of s, there are edges from s to u_i with the label of the element name.

A path in the schema graph is a cycle-free sequence of edges. The label of a path, *label(p)* is the "." separated concatenation of edges of p, analogeously to the definition of path labels in a database. Note, because not every edge has a label, the size of a path label can be shorter than the number of edges in the path. The set of paths between two nodes u, v in the schemagraph is denoted by *paths(u, v)*; as a special case, if u is the root node, we allow the unary function *paths(v)* as an abbreviation to denote all paths to v starting with

the root node. For a set of paths P, *labels(P)* denote the set of labels associated with the paths in P.

A path $p = (e_1, e_2, \ldots, e_n)$ is called *elementary* or *e-path*, if for $1 \leq i < n$, e_i is without label and e_n has a label $l \in \mathcal{L}$. I. e. a path is an e-path, if only the last edge in the sequence has a label and all other edges either are *isa* edges or have a substitution group as target.

The formal representation of a schemagraph $S = (T \cup SG, E^S_{\mathcal{L}} \cup E^S_{isa}, r_S)$ consist of a set of types T and substitution groups SG, edges representing references between types or substitution groups $E^S_{\mathcal{L}}$, a partial order between types E^S_{isa}, and a root type r_S.

We now can define the relationship between an XML database and an XML schema by the mapping $inst$, which maps a type of a schema to objects in the database. Let S and d be a schema and a database as defined before, then $inst : T \mapsto O$, such that for $t \in T$: *inst(t)* $= \{o \in O | \exists p \in$ *paths(t)* and $o \in$ *objects(label(p))*$\}$. Also, if $o \in O$ is instance of type t, we say o has type t, i.e. type(o) $= t$.

The instances of a type t therefore can be determined by taking all paths from the schema root to type t and by traversing the database along these paths.

Proposition 1 *Given a schemagraph S as before. For types $t, t' \in T$ of S and t isa t': labels(paths(t)) $\subseteq$ labels(paths(t')).*

This yields to an interesting relationship, also known from object oriented data models. If *t isa t'* in the schema graph, then *inst(t)* $\subseteq$ *inst(t')*. The proof is omitted here; only some examples are given to illustrate this relationship.

Example 2 *inst(*`BOOK`*)* $\subseteq$ *inst(*`PUBLICATION`*), because:*

a. *labels(paths(*`BOOK`*))* = $\{$*university.library.books.book*$\}$ = P_1
inst(`BOOK`*)* = $\bigcup_{p \in P_1}$ *objects(p)* = $\{$*36, 37*$\}$

b. *labels(paths(*`PUBLICATION`*))* =
$\{$*university.researcher.publications.publication,*
university.researcher.publications.article,
university.researcher.publications.monograph,
university.library.books.book$\}$ = P_2
inst(`PUBLICATION`*)* = $\bigcup_{p \in P_2}$ *objects(p)* = $\{$*7, 18, 24, 36, 37*$\}$

In many cases, one is interested in knowing if a certain database d is "compliant" with a schema S. This brings up the notion of *validity* of a database d with respect to a schema S. A database d is *valid with respect to schema S*, if all of the following hold: (i) $\forall o \in O$: *type(o)* is defined, (ii) *type*$(r_d) = r_S$, (iii) $\alpha(o) \in$ domain(`STRING`) and (iv) if $e_d = (o, l, o') \in E^D_{\mathcal{L}}$, then there is an e-path from *type(o)* to *type(o')* with label l in S. If d is valid w.r.t. schema S, we say, d is an *instance* of schema S. The database d from Figure 3 therefore is an instance of the schema in Figure 4.

3 Queries

In this section we present the notion of strict and concept based queries to access data in a database. Strict queries basically represent the well known access to XML data; they are interpreted under a "rigid matching" semantics as e.g. described in [KS01]. This semantics is straightforward and used in multiple XML based languages.

On top of strict queries we define concept based queries. Concept based queries allow to specify type information in a query and in particular restrict the possible matchings for a variable according to that type.

The semantics of concept based queries is described by means of one or multiple strict queries. That is, every concept based query can be expressed by a set of strict queries. Hence, we do not add expressive power to the query language, but we reduce (sometimes drastically) the structural complexity of queries and the amount of typing a user has to do in order to write complex queries.

We use a two step process to evaluate concept based queries. In the first step, a concept based query is "completed" into one or multipe strict queries. In a second step the resulting strict queries are evaluated. This two step proceeding allows, in a multitier architecture, client based completion of queries and provides a possibility for a user to select only some of the possible strict queries. If there is no completion, this can be determined directly at client side, without access to a database server.

3.1 Strict queries

Queries are represented by means of query graphs [KS01]. However, our definition of queries differs from the one in [KS01] in that we consider all query variables as typed ones. Formally, a *query* is a 3-tuple $q = (V, E^q_{\mathcal{L}}, r_q)$ where V is a finite set of typed variables, $E^q_{\mathcal{L}}$ is a set of labeled edges and r_q is the root of the query. For a variable v of type t, we write $t[v]$. If t equals the most general type $\texttt{ANY}$, we also simply write v instead of $\texttt{ANY}[v]$. If all variables in q are of type $\texttt{ANY}$, q is called *any-typed*.

Let $q = (V, E^q_{\mathcal{L}}, r_q)$ be a query and $d = (O, E^d_{\mathcal{L}}, r_d, \alpha)$ be a database that is an instance of a schema S. A *matching* of q w.r.t. d is a mapping $\mu : V \to O$ that satisfies the constraints imposed by q. We consider the following *constraints*:

- The *root constraint (rc)* requires the root of q to be mapped to the root of d.
- The *edge constraint (ec)*, written as ulv, where ulv is an edge in q. The ec ulv is satisfied by the mapping μ if d has an edge labeled with l from $\mu(u)$ to $\mu(v)$.
- The *concept constraint (cc)*, written as $c[u]$ where $u \in V$ is satisfied, if $\mu(u) \in inst(c)$ for a type c in S.

A *rigid matching* for a query q is a matching that satisfies the rc, all the ec's and all cc's of the query. The set of all rigid matchings of q w.r.t. d as instance of S is denoted as

$Mat_{d_S}(q)$. For the special case of any-typed queries, we can omit the schema (because type ANY) belongs to every schema) and write $Mat_d(q)$ for the set of all matchings of a query. Actually, $Mat_d(q)$ reflect the usual semantics of queries [KS01].

Example 3 *Figure 5 shows three example queries: a) asks for all* `publication` *objects contained in* `publications` *of a* `researcher` *in a* `university` *together with the* `researcher` *name. Query b) is similar, but specifies* `book` *objects instead of* `publication` *objects. The query c) is also similar to a) but specifies by means of z_1 and z_2 that we are only interested in researchers, that have at least two* `publication` *objects. The grey nodes indicate the root of the query, respectively.*

Actually, related to our example database d, only for query a) $Mat_d(q)$ is not empty. $Mat_d(q)$ contains one element which is described by the following binding $(u = 0, v = 1, w = 3, x = 4, y = 6, z = 18)$. Query b) and c) do not have any rigid matchings because related to b) the `publications` *node with id 6 has no outgoing edge with label* `book` *and related to c) this* `publications` *node only has one* `publication` *sub node but not a second one.*

Evidently, the reasons why $Mat_d(q)$ is empty for queries b) and c) are different. If we assume, that researcher "Smith" writes another publication in the future, query c) may be satisfiable under this new database. Query b) however won't be satisfiable at any point in time as long as we assume, that the database is an instance of our example schema. These observations bring us to the notion of typing.

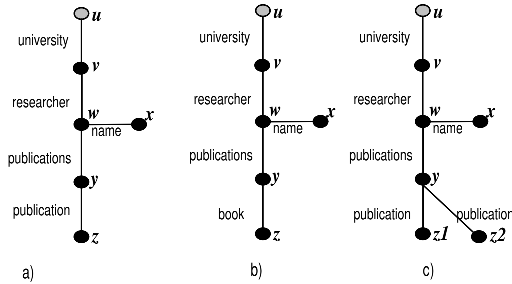


Figure 5: Some example queries

3.2 Typing

Type information is useful to determine if queries have a matching at all. By means of type information, query optimization is possible, that is, variables can be bound to their most specific types and furthermore, invalidly typed queries can be rejected. Checking query types can be done without the database only by means of a schema. Type checking usually is done by means of DTDs or by mechanisms to exploit structure from the semistructured database [BDFS97, GW97].

In this section we present the notion of typing related to the schemagraph and describe how subtype relationships are exploited in determining query types.

Let $q = (V, E^Q_{\mathcal{L}}, r_q)$ be a query and $S = (T \cup SG, E^S_{\mathcal{L}} \cup E^S_{isa}, r_S)$ be a schema graph.

A *type assignment* of q w.r.t. S is a mapping $\tau : V \to T$ that satisfies the *type constraints*

imposed by the query q. We consider the following constraints

- The *schema root constraint (src)* requires the root node of q to be mapped to the node `ROOT` in S.
- The *elementary path constraint (epc)* written as $u * lv$. The constraint $u * lv$ holds w.r.t. τ, if for edge (u, l, v) in q, an e-path from $\tau(u)$ to $\tau(v)$ with label l in S exists.
- The *concept constraint (cc)*, written as $c[u]$. The cc c[u] holds w.r.t. τ if $\tau(u)$ *isa* c in S.

A query q is called *strictly typed (or strict)* with respect to S, if there is a type assignment that satisfies the src, all epc's and all cc's for q.

Proposition 2 *For a given strict query q, a schema S and a database d, if q has a non-empty rigid matching related to d, q also has a valid type assignment related to S. Furthermore, if query q does not have a valid type assignment, then q only has empty rigid matchings.*

The semantics of strictly typed queries is the rigid matching of the query. Queries that are not strictly typed always have an empty rigid matching. That is, the notion of strictness yields a sufficient criteria for queries with empty matchings.

Example 4 *Let us consider again the queries from Figure 5. Queries a) and c) are strictly typed related to the schema in Figure 4 whereas query b) is not strict w.r.t. to this schema.*

3.3 Concept based queries

Concept based queries are queries that do not have to satisfy the schema root constraint as required for strict queries. In particular, a *concept based* query is a query, that satisfies the schema root constraint if r_q is defined, the epc's for all edges in q and the cc's for all nodes in q. The following example illustrates some concept based queries and lists the possible type assigments for these queries.

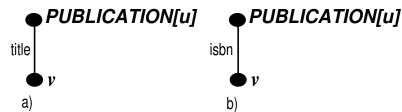


Figure 6: Concept based queries

Example 5 *Consider the queries in Figure 6. Query a) asks for all publications objects with its title information; b) requests publications together with their isbn. The type assignments T_a and T_b for queries a) and b) are given in the following tables:*

T_a	u	v
	PUBLICATION	STRING
	ARTICLE	STRING
	BOOK	STRING
	MONOGRAPH	STRING

T_b	u	u
	BOOK	STRING
	MONOGRAPH	STRING

Note, in the example before, valid type assignments eventually associate a type with a variable, that is a subtype of the type specified in the query. E.g. query b) specifies, that variable u is of type `PUBLICATION`. All type assignments found, though, associate either `BOOK` or `MONOGRAPH` with u. These assignments reflect pretty much the semantics of type instances introduced in section 2. The instances of a type t are all those objects in the databases, which are reachable by some paths that start at the root and ends in t. Actually, there are only paths going via `BOOK` and `MONOGRAPH` which have this property.

The remaining of the paper deals with the transformation of concept based queries into strict queries.

3.4 Processing of concept based queries

Concept based queries make use of schema information. Basically, a concept based query represents one or multiple strict queries; in particular, the semantics of a concept based query is defined in terms of one or multiple strict queries that are so called *completions* of a concept based query.

Figure 7 illustrates the processing of concept based queries. A concept based query is completed first into a set of strict queries. This set of completed strict queries can be further restricted by a user and then is evaluated against the database. The result of the concept based query is the union of all strict queries.

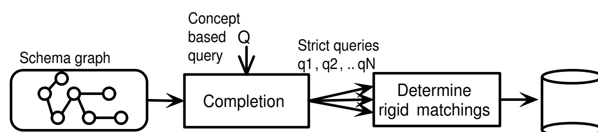


Figure 7: Processing of concept based queries

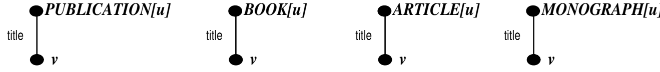
More formally, given a concept based query $q = (V_q, E^q_{\mathcal{L}}, r_q)$ and a set of valid type assignments T for q w.r.t. schema S, the set of *t-assigned* queries $Assign(q)$ is defined as follows:

Assign(q) = $\{q' | q'$ is concept based query, where every node u in q is replaced by $\tau(u)$ in q' for $\tau \in T\}$.

For a t-assigned query, every node u in q is assigned with a type t such that t isa t' and t' is the type of the variable u in query q. For the concept based queries in Figure 6 query b) will have two t-assigned queries; one where u is assigned type `BOOK` and one where u is assigned `MONOGRAPH`.

Based on t-assigned queries we now can define the root path completion of a query q. The *root path completion* for a query q is a query q', where every node u with no incoming edge is replaced by a path from the schema root to the type of u. The set of all root path completions for a query q is called root-compl(q). That is:

a) Assign(PUBLICATION[u].title[v])



b) Root completions

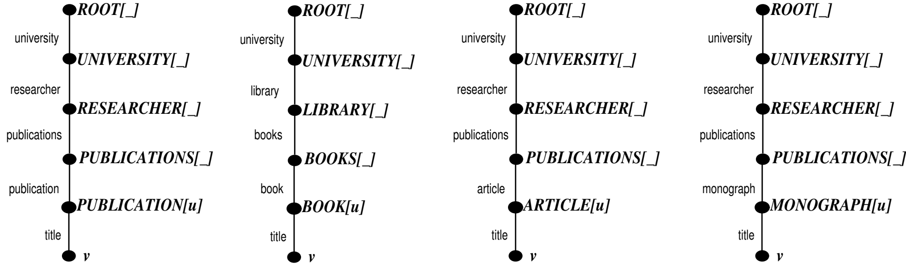


Figure 8: Completions of queries in Figure 6

root-compl(q) = $\{q'|q'$ contains q as a subgraph and for all variables $t[u]$ in q, if $t[u]$ has no incoming edge, then a path $p \in paths(type(u))$ is added to $q'\}$

Then, for a concept based query q, the completion *compl(q)* is defined as:

comp(q) = $\{q'|q' \in$ *root-comp(q'')* where $q'' \in$ *Assign(q)*$\}$

Example 6 *Consider query a) in Figure 6. This query can be express in a linearized form as a conjunctive query* `PUBLICATION[u] and [u].title[v]`*, see e.g. [KKS92]. The completion of this rather concise query is shown in Figure 8. There are four completions for the query, and the union of all matchings restricted to* u *and* v *is given as follows:*

	u	v	$\alpha(v)$
	18	*19*	*"XML ..."*
	36	not covered by example	
	37	*38*	*"Databases .."*

	u	v	$\alpha(v)$
	7	*8*	*"Flex Queries"*
	24	*25*	*"Basics ..."*

By means of a query that requests objects of a certain type, we can formulate rather concise queries. Actually, users who only have some idea about the concepts represented in a database are able to query data, even if they do not know how different XML elements are related.

Furthermore, if a couple of database schemas use common (general) concepts and share at least part of the *isa* hierarchy, queries are possible that access different databases. Another aspect related to concept based queries which exploit subtype / supertype relationships is that queries which simply ask for objects of a certain type, are more robust against schema changes that change the relationships between objects.

4 Related work

This section covers related work. There are a couple of query languages for semistructured or XML data. Two kinds of approaches that ease querying can be distinguished: Query languages, which provide some flexibility in that the user does not have to specify all details of the database structure, and approaches which direct the user when specifying his query by means of so called "data guides".

XPath [XPa99] and XQuery [XQu01] provide some flexibility by means of wild card characters "*" and "//" which describe all direct subelements or all descendents of the current node. XPath and XQuery, however, do not allow schema expressions in queries.

[KS01] uses the notion of flexible queries; a flexible query is a query graph; an edge in the graph is interpreted by (multiple) complete paths in the database. Paths are treated bidirectionally. This approach is quite powerful and releases the user from many details of the database structure. The focus in [KS01] is on "abbreviation" of aggregation relationships whereas our focus is on exploiting the specialization relationship available in the schema.

[May01] proposes XPathLog, a query language based on XPath but tailored towards the use within integration scenarios. XPathLog uses concepts from an ontology within query expressions but these ontologies are general ones and not directly related to the structure in the XML data. Also, the completion step of our approach does not rely on the underlying data source but only needs the schema.

Two data guide approaches are presented in [BDFS97] and [GW97]. Both approaches start from the XML data and derive structural information. Generalization relationships are not considered, because these are not explicitly available in the data.

5 Summary

In this article we have proposed concept based queries. This approach can be seen as an extension to existing strict queries. By using concept names we can formulate quite complex though rather concise queries. Compared to a wild card operator, using concept names is more intuitive and the result is easier to understand.

The two step process for evaluating queries, allows a user to explicitly select completed strict queries derived from concept based ones. This avoids access to objects not relevant to the user. Furthermore, because only the schema is necessary, the completion can be done at client side without using the eventually slow connection to the database server.

In particular, we have made use of generalization relationships available in an XML schema. This allows to access all instances of a concept, no matter, if they are direct instances of a requested concept or if they are instances of subconcepts.

The formal framework used will allow us more research on the subject. In particular, we will consider how we could add more flexibility to the query language (e.g. omitting edges in the query graph).

References

[Abi97] S. Abiteboul. Querying Semistructured Data. In *ICDT*, pages 1–18, 1997.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[BCN92] C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design*. Benjamin Cummings, 1992.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In *ICDT*, pages 336–350, 1997.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *SIGMOD*, 1994.

[Cat96] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann, San Francisco, CA, 1996.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML Data. *IEEE Data Engineering Bulletin*, 1999.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of the 23th VLDB Conference*, Athens (Greece), 1997. Morgan Kaufmann.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In *SIGMOD*, 1992.

[KS01] Y. Kanza and Y. Sagiv. Flexible Queries over Semistructured Data. In *PODS*, 2001.

[May01] W. May. A Framework for Generic Integration of XML Data Sources. In *Int. Workshop on Knowledge Representation meets Databases (KRDB 2001)*, Rome, Italy, 2001.

[ML01] P.J. Marron and G. Lausen. On Processing XML in LDAP. In *Proc. of the 27th VLDB Conference*, Roma, Italy, 2001.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. 11th Int. Conf. on Data Engineering*, pages 251–260, Taipei, 1995.

[XPa99] W3C, *XPath Specification*, 1999. `http://www.w3.org/TR/xpath`.

[XQu01] W3C, *XQuery 1.0: An XML Query Language*, 2001. `http://www.w3.org/TR/xquery/`.

[XSc01] W3C, *XML Schema - Part 0 to Part 2*, 2001. `http://www.w3.org/TR/xmlschema-{0|1|2}/`.