SQL4X: A Flexible Query Language for XML and Relational Databases

Sara Cohen, Yaron Kanza, and Yehoshua Sagiv [⋆]

The Hebrew University, Institute of Computer Science, Jerusalem 91904, Israel

Abstract. SQL4X, a powerful language for simultaneously querying both relational and XML databases is presented. Using SQL4X, one can create both relations and XML documents as query results. Thus, SQL4X can be thought of as an *integration language*. In order to allow easy integration of XML documents with varied structures, SQL4X uses *flexible semantics* when querying XML. SQL4X is also a powerful query language. It can express quantification, negation, aggregation, grouping and path expressions.

$\mathsf{Datalog}_{4x}$ and $\mathsf{Tree\text{-}Datalog}_{4x}$, extensions of $\mathsf{Datalog}$, are defined as elegant abstract models for SQL4X queries. Query containment is characterized for many common classes of SQL4X queries. Specifically, for $\mathsf{Datalog}_{4x}$ queries, a complete characterization of containment of conjunctive queries and of unions of queries is presented. Equivalence of $\mathsf{Datalog}_{4x}$ queries under bag-set semantics is also characterized. A sufficient condition for containment of $\mathsf{Tree\text{-}Datalog}_{4x}$ queries is presented. This condition is shown to be complete for a large class of common queries.

1 Introduction

The increasing popularity of XML has inspired the proposal of many XML query languages. Some of the better known languages are XQuery [CCF$^+$01], Lorel [AQM$^+$96,GMW99], XML-QL [DFF$^+$98,DFF$^+$99] and XSL [Cla99]. The language presented in this paper, called SQL4X, has several important features lacking in other languages.

The design philosophy of SQL4X is to extend SQL and relational database technology in order to support XML. The syntax of SQL4X is very similar to standard SQL syntax. In fact, SQL4X is defined to be SQL with a few additional features. This allows easy and natural implementation of SQL4X over a relational database, in a way that automatically utilizes advanced database features, such as indexing and query optimization. We have already implemented SQL4X with little effort over the Oracle Database System.

In many common scenarios, it is necessary to integrate relational and XML databases. Therefore, in SQL4X we allow simultaneous querying of both relations

[⋆] This work was supported in part by Grant 9481-3-00 from the Israeli Ministry of Science.

and XML documents. Similarly, our queries can create both relations and XML documents as query results. Thus, SQL4X can be thought of as an *integration language.* In order to allow easy integration of XML documents with varied structures, SQL4X uses *flexible semantics,* as defined in [KS01], for querying XML. In order to deal gracefully with partial information, queries in SQL4X return either an empty table or an empty XML fragment when the conditions of the query cannot be satisfied.

Elegant abstract models, such as Datalog, have been defined for SQL queries. This has allowed scores of researchers to tackle problems such as containment, equivalence and view usability of classes of SQL queries. Such results have inspired many optimization techniques which, in turn, allow relational databases to be efficiently accessed [CM77,ASU79,JK83,LMSS93,CKPS95]. We show how techniques for query containment of SQL queries can be extended to containment of SQL4X queries. Thus, known results about containment of relational queries can be applied to SQL4X queries.

Since SQL4X is simply an extension of SQL and the two languages are very similar, our language is easy to learn for anyone familiar with both SQL and XML. Furthermore, we expect that SQL4X will naturally evolve as the SQL standard changes and develops. In addition to the features stated above, SQL4X is also a powerful query language. It can express negation, quantification, aggregation and grouping.

In Section 2 our data model is presented. Section 3 presents the SQL4X language, by examples. In Section 4 we present results for containment of queries that generate relations, and in Section 5 containment of queries that generate documents is discussed. Section 6 compares SQL4X to other query languages and concludes.

2 Data Model

Using SQL4X, one can simultaneously query data sources in two formats: tables and documents. Tables are represented as relations and are not discussed further. Documents are represented using rooted labeled ordered trees. As in the OEM data model, we assume that each node has a unique id[1]. Since we will not assume that documents have associated schemas, in this paper trees representing documents do not follow ID and IDREF links.

Let $\mathcal{L}$ be a set of labels. Formally, an *ordered rooted labeled tree* is a 4-tuple $t = (N, E, r, \prec)$ where *(1)* N is a set of nodes; *(2)* $E \subseteq N \times N \times \mathcal{L}$ is a set of labeled edges that gives rise to a tree-like structure; *(3)* $r \in N$ is the root; and *(4)* $\prec$ is a partial order on N, which defines a complete order on sibling nodes. We say that a node n is *complex* if it has at least one outgoing edge. Otherwise, n is *atomic.* We denote the atomic nodes of N by N_a.

[1] We do not assume that the unique id is actually present in the document. Instead, our system generates unique ids as needed. There is no relationship between system generated ids and values of ID attributes of elements. Id values can be generated, for example, using XPointer [DMD01].

Table 1. Functions defined on nodes which can be used in SQL4X queries.

Function	Value
`oid(`n`)`	object id of n
`label(`n`)`	label of n's incoming edge, or null if n is the root
`data(`n`)`	the set of subtrees below n (i.e., the document fragment surrounded by n)
`text(`n`)`	the concatenation of the strings in the atomic nodes of `data(`n`)`
`index(`n`)`	(the number of siblings with the label `label(n)`preceding n) + 1
`doc(`n`)`	the document in which n appears
`is_att(`n`)`	true if n is an attribute, false otherwise
`path(`n`)`	the labels on the path from the root to n, or null if n is the root
`path(`n`,`m`)`	the labels on the path between n and m, if n is above m; the labels on the path between m and n, if m is above m; otherwise, null
`distance(`n`,`m`)`	the number of edges on the path from n to m; the value is positive (negative) if m is below (above) n; the value is null if n and m are not connected

Let $\mathcal{A}$ be a set of strings, including the empty string ϵ. A *document* is a pair $x = (t, \alpha)$ where $t = (N, E, r, \prec)$ is a tree and $\alpha \colon N_a \to \mathcal{A}$ is a function that maps the atomic nodes in N to strings. A *database* is a pair $\mathcal{D} = (R, X)$, where R is a set of relations and X is a set of documents.

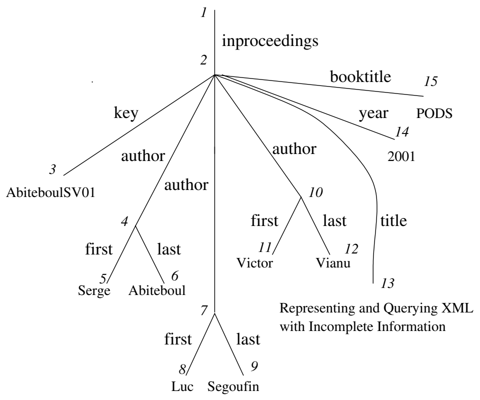
Functions defined on document nodes, and which can be used in SQL4X queries, are presented in Table 1.

Example 1. In Figure 1, a document fragment and corresponding tree are depicted. The following are some values of the functions in Table 1 on nodes in the tree from Figure 1: *(1)* `text(10) = Victor Vianu`, *(2)* `label(10) = author`, *(3)* `data(10) = <first>Victor</first><last>Vianu</last>`, *(4)* `index(10) = 3`, *(5)* `path(10) = inproceedings.author` *(6)* `path(11,2) = author.first` and *(7)* `distance(11,2) = -2`.

3 Language Definition

SQL4X is an extension of SQL. We assume that the reader is familiar with SQL. Thus, we only present the features that are not part of SQL. Most of our presentation will be by example. The examples are mainly inspired by [FSW99] which presents examples of queries in several XML query languages.

An SQL4X query begins with one of the following two declarations: *(1)* `QUERY AS RELATION` or *(2)* `QUERY AS DOCUMENT`. If Declaration 1 is present, the result of the query is a relation, as in a standard SQL query. We call such queries *relation-generating queries*. If, instead, Declaration 2 is present, the result is an XML document. Such queries are called *document-generating queries*. For



```
<inproceedings
        key = "AbiteboulSV01">
   <author><first>Serge</first>
           <last>Abiteboul</last>
   <author>
   <author><first>Luc</first>
           <last>Segoufin</last>
   </author>
   <author><first>Victor</first>
           <last>Vianu</last>
   </author>
   <title>Representing and Query-
          ing XML with Incomplete
          Information </title>
   <year>2001</year>
   <booktitle>PODS</booktitle>
</inproceedings>
```

Fig. 1. Labeled tree corresponding to the document fragment.

relation-generating queries, only the `FROM` clause is interpreted differently from a standard SQL query. In document-generating queries, the `SELECT` clause is also interpreted in a non-standard fashion.

3.1 `FROM` Clause Extensions

We allow the `FROM` clause of an SQL4X query to contain table names, possibly with aliases. As in SQL, including a relation $R(a_1, \ldots, a_n)$ in the `FROM` clause of a query implicitly defines *field variables* $a_1, \ldots, a_n$ that range over the values in the columns $a_1, \ldots, a_n$, respectively, in the table R. We allow additional constructs in the `FROM` clause that define *node variables* that range over nodes in an XML document tree. The construct

$$v\ l\ \texttt{of}\ \text{'}\textit{fname-exp}\text{'}$$

defines a node variable v that ranges over nodes with the incoming label l in files *matching 'fname-exp'* (with matching defined by the SQL function `LIKE`). Similarly, the construct

$$v\ l\ \texttt{of}\ u$$

defines a node variable v that ranges over nodes with the incoming label l that are connected by a path to the node variable u. Note that this implies that u is either an ancestor or a descendant of v (or $u = v$) in a document tree. The functions in Table 1 can all be used in an SQL4X query.

Example 2. Suppose that the document `pods01.xml` contains many document fragments of the type depicted in Figure 1. Suppose also that the relation `Affiliation(name, institution)` is a table that contains information about the affiliation of prominent database researchers. The following query returns a list of pairs of INRIA authors and titles of their articles as described in `pods01.xml`.

```
QUERY AS RELATION
SELECT text(a) AS author, text(t) AS title
FROM i inproceedings of ‘pods01.xml’, a author of i, t title of i,
     Affiliation
WHERE name = text(a) and institution = ‘INRIA’
```

Including the relation `Affiliation` in the `FROM` clause automatically defines the variables `name` and `institution`. Note that the author and title pairs will in fact be such that the author wrote an article with the corresponding title. This is ensured by requiring that there be a path between both i and a, and i and t.

The reader will also observe the simple fashion in which relations and documents were simultaneously queried. XML documents, such as `pods01.xml` can be downloaded from the DBLP website [Ley], and then integrated with information residing in a relational database.

Example 3. Consider a database containing many XML documents `pods01.xml`, `pods00.xml`, `pods99.xml`, etc., each of which contains information about articles published in the PODS conference of the corresponding year. We can find the names of the documents in which Vianu published an article, along with the titles of the articles, with the following query:

```
QUERY AS RELATION
SELECT doc(l) AS docname, text(t) AS title
FROM i inproceedings of ‘pods__.xml’, l last of i, t title of i
WHERE text(l) = ‘Vianu’
ORDER BY text(t)
```

It is of interest to note that although elements with label `last` do not appear directly below elements with labels `inproceedings`, the query will still retrieve the required data. This is true since the query only requires the existence of a *path* between i and l. This allows querying of documents even when the exact structure of the document is not completely known.

3.2 `SELECT` Clause Extensions

Relation-generating SQL4X queries contain `SELECT` clauses exactly as in SQL. We discuss the interpretation of the `SELECT` clause in a document-generating query. In many ways our extensions to the `SELECT` clause are not significantly different from constructs in query languages such as XML-QL [DFF$^+$98] and XQL [RLS98]. However, our language differs in being a rather natural extension of SQL. We will also demonstrate the use of SQL constructs such as quantification and aggregation in SQL4X.

Basic Rules. A `SELECT` clause has the same form as an XML document fragment. Specifically, it may contain tags, attributes and SQL expressions that evaluate to values. Everything written outside of a tag or as the value of an attribute is assumed to be an expression that must be computed.

Example 4. Recall the query in Example 2. Suppose we are interested in the same result, but as an XML document. We will retrieve in addition the year of publication of the articles. This is almost accomplished with the query below. The query returns the desired result, but without a root document element.

```
QUERY AS DOCUMENT
SELECT <pair><author> data(a) </author>
             <title year = data(y)> data(t) </title> </pair>
FROM i inproceedings of ‘pods01.xml’, a author of i, t title of i,
     y year of i, Affiliation
WHERE name = text(a) and institution = ’INRIA’
```

As explained above, `data(a)` and `data(t)` are interpreted as expressions, since they are outside of a tag. Similarly, `data(y)` is interpreted as an expression since it is the value of an attribute. This query produces a string (or document) consisting of many fragments of the following form:

```
<pair> <author><first> Serge </first>
                <last> Abiteboul </last> </author>
       <title year = ‘‘2001’’> Representing and Querying XML
       with Incomplete Information. </title> </pair>
```

Embedding Queries in a Document. The query in Example 4 did not actually produce a legal XML document since there was no *root tag* in the document, i.e., a tag that enclosed all of the information in the document. Creating such an enclosing tag is quite simple. In an SQL4X query, standard SQL clauses can be placed between XML elements. The result of evaluating such a query is obtained by evaluating the SQL-like portion of the query and then pasting its result between the XML tags.

Example 5. The following query will produce a list of all the titles and authors in `pods01.xml`.

```
QUERY AS DOCUMENT
<bib> <titles> (SELECT DISTINCT <title> data(t) </title>
                FROM t title of ‘pods01.xml’)
      </titles>
      <authors> (SELECT DISTINCT <author> data(a) </author>
                 FROM a author of ‘pods01.xml’)
      </authors> </bib>
```

As in a standard SQL query, the `DISTINCT` keyword will eliminate duplicate document fragments.

Subqueries in the `SELECT` Clause. Standard SQL allows subqueries in the `SELECT` clause. These subqueries must return single values. Similarly, we allow subqueries in the `SELECT` clause, and each such subquery returns a document fragment. The semantics of query evaluation with such subqueries is exactly as in SQL, i.e., nested-loop semantics.

3.3 Advanced Features

There are several advanced features in SQL4X that give additional expressive power beyond the constructs in SQL. This section is devoted to describing these features.

Aggregation. Since all SQL constructs can be used in SQL4X, standard aggregate functions from SQL can be used in SQL4X queries. It is possible to use `GROUP BY` and `HAVING` clauses in an SQL4X query. An additional aggregate function is available in SQL4X. The `ALL` aggregate function concatenates the set of document fragments to which it is applied. Use of the `ALL` aggregate function can sometimes eliminate the need for a subquery.

Example 6. The query below finds for each author in `pods01.xml`, all the articles that she has written. Note that it is possible to formulate an equivalent query which uses a subquery and does not use the `ALL` aggregate function.

```
QUERY AS DOCUMENT
<bib>
SELECT <inproceedings><author> data(a) </author>
                     ALL(<title> data(t) </title>)
       </inproceedings>
FROM i inproceedings of ‘pods01.xml’, a author of i, t title of i
GROUP BY data(a)
</bib>
```

The function `ALL(DISTINCT ...)` can be used to eliminate duplicates in the function arguments.

Integrating Documents with Different Structures. The construct for binding node variables, v l `of` u, only requires the existence of a path between v and u. This greatly differs from other common XML query languages which require exact matching of the structure. In other languages, a regular expression can be used to simulate this flexible matching ability. Even then, it may be difficult to formulate queries that rely heavily upon flexible matchings.

Example 7. Suppose that we have a database that holds documents describing courses, students taking courses and professors teaching courses. Some documents were written by students, some by professors and some were written for course needs. Yet, all these documents use the element names `student`, `course`, `teacher`, `sname`, `cname` and `tname`. However, the documents are structured in different hierarchies depending on the point of view of the creator. It is possible to get a single view of all these documents, with the course at the top of the hierarchy, using the following query.

```
QUERY AS DOCUMENT
<info>
SELECT <course>
        <name> data(nc) <name>
        <teachers> ALL(<teacher> data(nt) </teacher>) </teachers>
        <students> ALL(<student> data(ns) </student>) </students>
       </course>
FROM c course of '%.xml', nc cname of c, t teacher of c,
     nt tname of teacher, s student of c, ns sname of student
GROUP BY data(nc)
<info>
```

Note that using `'%.xml'` will cause the query to be applied to all documents in the database. If desired, one can restrict the structure of the documents that are used to create the results. For example, it is possible to require that the queried document has `teacher` hierarchically above `course` by adding `distance(t,c) > 0` to the `WHERE` clause of the query. Adding `distance(t,c) = 1` to the `WHERE` clause restricts course elements to be children of teacher elements.

Negation and Label Variables. SQL4X can express queries with negation. Negation in SQL4X is similar to negation in SQL and is expressed with the `not exists` or `not in` constructs preceding a subquery.

SQL4X has *labels variables*. Label variables are variables that are bound to labels. Labels variables are useful for expressing path expressions. Due to a lack of space we will not discuss negation and label variables further.

4 Relation-Generating Queries

Relation-generating queries are similar to standard SQL queries. The important addition is the ability to query XML documents, in addition to relations. We show that it is possible to use classical techniques from relational queries to determine containment for such queries. We start by considering important restricted classes of SQL4X queries. Results from these classes can be built upon to derive results for more general queries. To simplify the presentation, we assume in this section that at most one document is queried in the `FROM` clause. Thus, in this section a database is a pair $\mathcal{D} = (R, x)$ where R is a set of relations and x is a single document. This restriction can easily be lifted to yield results for queries with any number of documents in the `FROM` clause.

4.1 Syntax and Semantics

We use an extended Datalog syntax, called $\mathsf{Datalog}_{4x}$, for our queries. Variables are denoted by u, v, w. A term, denoted s or t, is a variable or a constant. Predicates symbols are denoted as p, q, r. A *relational atom* has the form $p(s_1, \ldots, s_k)$ where s_i are terms and p is a predicate of arity k. We sometimes use $p(\bar{s})$, where

$\bar{s}$ is a tuple of terms, as an alternative notation. A *label atom* has the form $\langle l \downarrow u \rangle$ where u is a variable and l is a constant, called the *incoming label* of u. A *path atom* has the form $\langle u \sim v \rangle$, where u and v are variables. Variables appearing in label atoms or in path atoms are called *node variables.* An *atom* is a relational atom, a label atom or a path atom.

We consider safe conjunctive $\mathsf{Datalog}_{4x}$ queries. Such queries have the form

$$\begin{aligned} q(\bar{s}) \leftarrow & p_1(\bar{s}_1) \ \& \ \ldots \ \& \ p_n(\bar{s}_n) \ \& \\ & \langle l_1 \downarrow u_1 \rangle \ \& \ \ldots \ \& \ \langle l_k \downarrow u_k \rangle \ \& \ \langle v_1 \sim w_1 \rangle \ \& \ \ldots \ \& \ \langle v_m \sim w_m \rangle \end{aligned} \tag{1}$$

where *(1)* all the variables in the head appear in the body, i.e., the variables in $\bar{s}$ appear among $\bar{s}_1, \ldots, \bar{s}_n, u_1, \ldots, u_k, v_1, \ldots, v_m, w_1, \ldots, w_m$; and *(2)* all the node variables have incoming labels, i.e., for all v_i (w_i) there is a variable u_j such that $v_i = u_j$ $(w_i = u_j)$. The variables in $\bar{s}$ are called the *distinguished variables.* We write a query as $q(\bar{s}) \leftarrow B(\bar{t})$ when the structure of the body is not important. We sometimes denote the above query by $q(\bar{s})$, or simply by q.

An *assignment* φ is a mapping of the terms in a query to constants, such that each constant is mapped to itself. Assignments are naturally extended to tuples and atoms. Satisfaction of a relational atom with respect to an assignment and a database is defined in the usual fashion. An assignment φ satisfies a path atom $\langle u \sim v \rangle$ with respect to a database containing the document x if $\varphi(v)$ is the object id of a node in x connected by a (possibly empty) path to a node with object id $\varphi(u)$. The assignment φ satisfies $\langle l \downarrow u \rangle$ with respect to a document x if $\varphi(u)$ is the object id of a node in x with the incoming label l.

A conjunctive query $q(\bar{s}) \leftarrow B(\bar{t})$ defines a relation $q^{\mathcal{D}}$ for a given database $\mathcal{D}$ as follows

$$q^{\mathcal{D}} := \{\varphi(\bar{s}) \mid \varphi \text{ satisfies } B(\bar{t}) \text{ w.r.t. } \mathcal{D}\}$$

We say that a query q is *satisfiable* if it returns a non-empty result for some database $\mathcal{D}$, i.e., $q^{\mathcal{D}} \neq \emptyset$.

Proposition 1. *A query q is satisfiable if and only if it does not contain label atoms $\langle l \downarrow u \rangle$ and $\langle l' \downarrow u \rangle$ such that $l \neq l'$.*

Satisfiability can be determined in linear time. In the sequel we only consider satisfiable queries.

Example 8. The following query is a $\mathsf{Datalog}_{4x}$ version of the query in Example 2. In order to conform with $\mathsf{Datalog}$ syntax, we have represented the function `text` by a binary relation, `textval`. Note that in order to correctly model this function as a relation, it must be assumed that the second field of the predicate `textval` is functionally dependent on the first field. However, a discussion of functional dependencies is beyond the scope of this paper.

$$\begin{aligned} \texttt{pairs}(ta, tt) \leftarrow & \langle \texttt{inproceedings} \downarrow i \rangle \ \& \ \langle \texttt{author} \downarrow a \rangle \ \& \ \langle \texttt{title} \downarrow t \rangle \ \& \\ & \langle a \sim i \rangle \ \& \ \langle t \sim i \rangle \ \& \\ & \texttt{textval}(a, ta) \ \& \ \texttt{affiliation}(ta, \text{`}\texttt{INRIA}\text{'}) \ \& \ \texttt{textval}(t, tt) \end{aligned}$$

4.2 Containment of $\mathsf{Datalog}_{4x}$ Queries

It is possible to characterize containment of conjunctive $\mathsf{Datalog}_{4x}$ queries using similar methods to those for conjunctive $\mathsf{Datalog}$ queries. Query containment is defined here in the usual fashion. We present necessary and sufficient characterizations for query containment.

Let $q(\bar{s})$ and $q'(\bar{s})$ be conjunctive $\mathsf{Datalog}_{4x}$ queries. A *homomorphism* from q' to q is a substitution θ of the variables in q' by terms in q that maps each distinguished variable to itself, such that *(1)* for all relational atoms a' in q', $\theta(a')$ is in q; *(2)* for all label atoms $\langle l \downarrow u' \rangle$ in q', $\langle l \downarrow \theta(u') \rangle$ is in q; and *(3)* for all path atoms $\langle u' \sim v' \rangle$ in q', either $\langle \theta(u') \sim \theta(v') \rangle$ or $\langle \theta(v') \sim \theta(u') \rangle$ is in q.

A query q is *contained* in a query q', written $q \subseteq q'$ if for all databases $\mathcal{D}$, it holds that $q^{\mathcal{D}} \subseteq {q'}^{\mathcal{D}}$.

Theorem 1. *Let $q(\bar{s}) \leftarrow B(\bar{t})$ and $q'(\bar{s}) \leftarrow B'(\bar{t}')$ be queries. If there exists a homomorphism from q' to q, then q is contained in q'.*

The condition in Theorem 1 is not necessary. It is possible that there is no homomorphism from q' to q and yet, q is contained in q'.

Example 9. Consider the queries

$$\begin{aligned} q(z) \leftarrow & a(z) \;\&\; \langle l \downarrow u \rangle \;\&\; \langle l \downarrow v \rangle \;\&\; \langle l \downarrow w \rangle \;\&\; \langle l \downarrow y \rangle \;\&\; \\ & \langle u \sim v \rangle \;\&\; \langle v \sim w \rangle \;\&\; \langle w \sim y \rangle \;\&\; \langle y \sim u \rangle \\ q'(z) \leftarrow & a(z) \;\&\; \langle l \downarrow u \rangle \;\&\; \langle l \downarrow v \rangle \;\&\; \langle l \downarrow w \rangle \;\&\; \langle l \downarrow y \rangle \;\&\; \\ & \langle u \sim v \rangle \;\&\; \langle v \sim w \rangle \;\&\; \langle w \sim y \rangle \;\&\; \langle y \sim u \rangle \;\&\; \langle v \sim y \rangle \end{aligned}$$

Clearly $q' \subseteq q$. By exhaustive search one can verify that it is impossible to create a tree with nodes u, v, w, y connected in the manner required by q without either u and w or v and y being connected. Intuitively, this follows since the path atoms create a cycle. Thus, q is equal to the union of the queries

$$\begin{aligned} q_1(z) \leftarrow & a(z) \;\&\; \langle l \downarrow u \rangle \;\&\; \langle l \downarrow v \rangle \;\&\; \langle l \downarrow w \rangle \;\&\; \langle l \downarrow y \rangle \;\&\; \\ & \langle u \sim v \rangle \;\&\; \langle v \sim w \rangle \;\&\; \langle w \sim y \rangle \;\&\; \langle y \sim u \rangle \;\&\; \langle u \sim w \rangle \\ q_2(z) \leftarrow & a(z) \;\&\; \langle l \downarrow u \rangle \;\&\; \langle l \downarrow v \rangle \;\&\; \langle l \downarrow w \rangle \;\&\; \langle l \downarrow y \rangle \;\&\; \\ & \langle u \sim v \rangle \;\&\; \langle v \sim w \rangle \;\&\; \langle w \sim y \rangle \;\&\; \langle y \sim u \rangle \;\&\; \langle v \sim y \rangle \end{aligned}$$

Note that $q_1 \subseteq q'$ and $q_2 \subseteq q'$. Thus, $q \subseteq q'$, even though there is no homomorphism from q' to q.

In order to present a necessary and sufficient condition for containment, we introduce some definitions. Consider a conjunction of label and path atoms $C = \langle l_1 \downarrow u_1 \rangle \;\&\; \ldots \;\&\; \langle l_k \downarrow u_k \rangle \;\&\; \langle v_1 \sim w_1 \rangle \;\&\; \ldots \;\&\; \langle v_m \sim w_m \rangle$. An assignment φ *exactly satisfies* C with respect to a document x if *(1)* φ satisfies all the atoms in C with respect to x, *and (2)* for all v_i and w_j, if $\langle v_i \sim w_j \rangle \notin C$ and $\langle w_j \sim v_i \rangle \notin C$, then φ does not satisfy $\langle v_i \sim w_j \rangle$ with respect to x. If there is a document x and an assignment φ such that φ exactly satisfies the conjunction of path and label atoms in q with respect to x, then we say that q is *exactly satisfiable.* Note that for q to be exactly satisfiable, it must contain path nodes $\langle w \sim w \rangle$ for all node variables w in q. We call such path atoms *self-path atoms.* Consider the query $q(\bar{s})$ in Equation 1. We say that $q'(\bar{s})$ is an *extension* of $q(\bar{s})$ if *(1)* q' contains exactly the same relational atoms and label atoms as q; and *(2)* q' contains all the path atoms in q and possibly additional path atoms with the node variables from q.

We say that q' is an *exactly satisfiable extension* of q if q' is an extension of q and q' is exactly satisfiable.

Theorem 2. *Consider queries q and q'. Then $q \subseteq q'$ if and only if for all exactly satisfiable extensions q_e of q, there is a homomorphism from q' to q_e.*

Corollary 1. *Consider queries q and q'. Suppose that q is exactly satisfiable. Then $q \subseteq q'$ if and only if there is a homomorphism from q' to q.*

Corollary 1 motivates the following question. What is the complexity of checking if a query q is exactly satisfiable? Suppose that q has k node variables. We can show that if q is exactly satisfiable, then there is an assignment φ and a document x with nodes N such that φ exactly satisfies the path and label atoms in q with respect to x and $|N| = k+1$. From this, follows that it suffices to check document trees with only $k+1$ nodes. However, there are an exponential number of such documents. Luckily, it turns out that this problem can be solved in polynomial time by a constructive proof mechanism. Specifically, there is an algorithm that given a query q creates, in polynomial time, a document x such that q exactly satisfies x, if such a document exists.

Theorem 3. *Given a query q, it is possible to decide in polynomial time if q is exactly satisfiable.*

Finally, we present an upper bound on the complexity of checking containment of $\mathsf{Datalog}_{4x}$ queries.

Proposition 2. *Determining containment of* $\mathsf{Datalog}_{4x}$ *queries is in Π_2^{P}.*

4.3 Extending Containment Results

In the previous subsection, the basic foundation for characterizing query containment has been laid. It is possible to extend these results for additional classes of queries. We give these results briefly.

Unions of Queries. The semantics of query evaluation is extended in the obvious fashion to unions of queries. Containment of union of queries is characterized by the following theorem.

Theorem 4. *Consider unions of queries $\bigcup_i q_i$ and $\bigcup_j q'_j$. Then $\bigcup_i q_i \subseteq \bigcup_j q'_j$ if and only if for all $q_i \in \bigcup_i q_i$ and for all exactly satisfiable extensions q_{i_e} of q_i, there is a $q'_j \in \bigcup_j q'_j$ for which there is a homomorphism from q'_j to q_{i_e}.*

Allowing unions of queries does not increase the complexity of query containment.

Corollary 2. *Determining containment of unions of* $\mathsf{Datalog}_{4x}$ *queries is in* Π_2^{P}.

Aggregate Queries and Bag-Set Semantics. We consider queries with the *all* and *all distinct* aggregate functions. A formal accounting of the semantics of aggregate $\mathsf{Datalog}_{4x}$ queries is not presented. See [NSS98,CNS99] for details about aggregate $\mathsf{Datalog}$ queries. We show how reasoning about queries with *all distinct* can be reduced to reasoning about non-aggregate queries (under set semantics). Similarly, reasoning about queries with the *all* function can be reduced to reasoning about non-aggregate queries under bag-set semantics. Queries are evaluated under *bag-set semantics*, if their input is a set, while their output is a bag (see [CV93]).

We consider aggregate queries with one of the following forms:

$$q(\bar{s}, \mathit{all}(y)) \leftarrow B(\bar{t}) \tag{2a}$$

$$q(\bar{s}, \mathit{all}(\mathit{distinct}\ y)) \leftarrow B(\bar{t}) \tag{2b}$$

where $\bar{s}$ are the *grouping variables*. The aggregate functions find for each satisfying assignment of $\bar{s}$, the set of all (distinct) satisfying assignments for y. Given an aggregate query q as defined above, the *core* of q, denoted $\breve{q}$, is the query $\breve{q}(\bar{s}, y) \leftarrow B(\bar{t})$. We consider aggregate queries having safe conjunctive $\mathsf{Datalog}_{4x}$ cores. We present an example and then characterize containment of aggregate queries.

Example 10. The following query groups authors with all the titles of their articles, as in Example 6.

$$\texttt{allPapers}(ta, \mathit{all}(\mathit{distinct}\ tt)) \leftarrow\ \langle \texttt{inproceedings} \downarrow i\rangle \ \&\ \langle \texttt{author} \downarrow a\rangle \ \&$$
$$\langle \texttt{title} \downarrow t\rangle \ \&\ \langle a \sim i\rangle \ \&\ \langle t \sim i\rangle \ \&\ \texttt{textval}(a, ta) \ \&\ \texttt{textval}(t, tt)$$

Proposition 3. *Given the queries* $q(\bar{s}, \mathit{all}(\mathit{distinct}\ y))$ *and* $q'(\bar{s}, \mathit{all}(\mathit{distinct}\ y))$, *then* $q \subseteq q'$ *if and only if* $\breve{q} \subseteq \breve{q}'$.

Proposition 4. *Given the queries* $q(\bar{s}, \mathit{all}(y))$ *and* $q'(\bar{s}, \mathit{all}(y))$, *then* $q \subseteq q'$ *if and only* $\breve{q}$ *is contained in* $\breve{q}'$ *under bag set semantics.*

Query containment under bag-set semantics is not known to be decidable [CV93,IR95]. Equivalence of $\mathsf{Datalog}$ queries under bag-set semantics has been characterized [CV93,NSS98,CNS99]. We extend these characterizations to deal with bag-set equivalence among $\mathsf{Datalog}_{4x}$ queries. We say that q is *isomorphic* to q' if there is a homomorphism θ from q' to q such that *(1)* θ is a bijection on the relational and label atoms of q; and *(2)* for all path atoms $\langle u \sim v\rangle$ in q, at least one of $\langle \theta^{-1}(u) \sim \theta^{-1}(v)\rangle$ and $\langle \theta^{-1}(v) \sim \theta^{-1}(u)\rangle$ is in q'.

Given sets of queries $Q = \{q_i\}_{i\in\mathcal{I}}$ and $Q' = \{q'_j\}_{j\in\mathcal{J}}$, we say that Q is isomorphic to Q' if there is a bijection γ from Q' to Q such that for all $q'_j \in Q'$, it holds that q'_j is isomorphic to $\gamma(q'_j)$.

Theorem 5. *Let q and q' be* $\mathsf{Datalog}_{4x}$ *queries. Then q is equivalent to q' under bag-set semantics if and only if the set of exactly satisfiable extensions of q is isomorphic to the set of exactly satisfiable extensions of q'.*

Corollary 3. *Bag-set equivalence of* $\mathsf{Datalog}_{4x}$ *queries can be decided in polynomial space.*

Corollary 4. *Let q and q' be exactly satisfiable* $\mathsf{Datalog}_{4x}$ *queries. Then q is bag-set equivalent to q' if and only if q is isomorphic to q'.*

As for Datalog queries, it is possible to identify classes of $\mathsf{Datalog}_{4x}$ queries for which equivalence can be decided in polynomial time. We say that a query q is *linear* if, in q, *(1)* no two relational atoms have the same predicate; *and (2)* no two label atoms have the same label, i.e., $l \neq l'$ for all $\langle l \downarrow u\rangle$ and $\langle l' \downarrow u'\rangle$ in q.

Theorem 6. *Bag-set equivalence of linear exactly satisfiable* $\mathsf{Datalog}_{4x}$ *queries can be decided in polynomial time.*

5 Document-Generating Queries

We consider document-generating queries. We present an abstract notation and discuss containment results. As in the previous section, we assume that at most one document is queried in a single query. As an additional simplification, we consider the output of a query to be unordered. Thus, documents are *unordered* rooted labeled trees, associated with a function mapping atomic nodes to values.

5.1 Syntax and Semantics

We use an extended Datalog syntax, called $\mathsf{Tree\text{-}Datalog}_{4x}$. As in $\mathsf{Datalog}_{4x}$, there are three types of atoms: relational atoms, label atoms and path atoms. A query is a rooted labeled tree, with (possibly empty) conjunctions of atoms attached to its edges and terms attached to its leaf nodes. Formally, a $\mathsf{Tree\text{-}Datalog}_{4x}$ query is a triple $q = (t, \Psi, \tau)$ where *(1)* $t = (N, E, r)$ is a rooted labeled tree; *(2)* Ψ associates each $e \in E$ with a conjunction of atoms, called the *condition* of e; and *(3)* τ associates each atomic node $n \in N_a$ with a term. We say that a variable v is *introduced* in an edge e if v appears in $\Psi(e)$ *and* v does not appear in $\Psi(e')$ for any e' above e in t. A query is *safe* if every variable associated with a leaf node n is introduced in an edge above n. A query is in *normal form* if no variable v is introduced in two different edges[2]. In the sequel we assume that all queries are safe and are in normal form.

$\mathsf{Tree\text{-}Datalog}_{4x}$ queries have semantics similar to nested-loops. Variables appearing in the condition associated with an edge get their values at the place where they are introduced. Assignments to atoms are extended to trees in the natural fashion. Thus, an assignment is applied to a tree by applying the assignment to all the conditions associated with the tree's edges and all the variables that are associated with the tree's leaves.

[2] Any query in non-normal form has an equivalent query in normal form.

```
QUERY AS DOCUMENT
<bib>
SELECT <inproceedings> <author> data(a) </author>
       (SELECT <title> data(t) </title>
          FROM i inproceedings of 'pods01.xml',
               ai author of i, t title of i
          WHERE text(a) = text(ai))
       </inproceedings>
FROM a author of 'pods01.xml'
</bib>
```

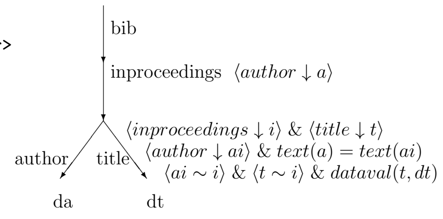

bib
inproceedings $\langle author \downarrow a\rangle$
$\langle inproceedings \downarrow i\rangle$ & $\langle title \downarrow t\rangle$
$\langle author \downarrow ai\rangle$ & $text(a) = text(ai)$
$\langle ai \sim i\rangle$ & $\langle t \sim i\rangle$ & $dataval(t, dt)$
author
title
da
dt

Fig. 2. An SQL4X query and its Tree-Datalog$_{4x}$ query.

Let $q = (t, \Psi, \tau)$ be a query. Suppose that n is a node in t. Then $q_{|n}$ is the query defined by the subtree of q rooted at n. Let $q = (t_q, \Psi, \tau)$ be a query and let $\mathcal{D}$ be a database. We denote the result of evaluating q with respect to $\mathcal{D}$ by $q^{\mathcal{D}}$ and it is obtained in the following fashion:

– If t_q consists only of single node r_q then $q^{\mathcal{D}}$ is the document $x = (t_x, \alpha)$ where $t_x = (\{r_x\}, \emptyset, r_x)$ and $\alpha(r_x) = \tau(r_q)$. Note that $\tau(r_q)$ must be a constant since q is safe.
– Otherwise, suppose that the outgoing edges from the root r_q are $e_1, \ldots, e_k$ where $e_i = (r_q, n_i, l_i)$. Let Φ_i be the set of satisfying assignments of $\Psi(e_i)$ into $\mathcal{D}$. Suppose that at least one of the sets Φ_i is non-empty. For each $\varphi \in \Phi_i$, we apply φ to q and then evaluate its subtree rooted at n_i on $\mathcal{D}$. Formally, we define X_i to be

$$X_i := \{(\varphi(q)_{|n_i})^{\mathcal{D}} \mid \varphi \in \Phi_i\}.$$

Let $N(X_i)$ be the set of nodes of documents in X_i. Similarly, we define $E(X_i)$, $r(X_i)$ and $\alpha(X_i)$. Then $q^{\mathcal{D}}$ is the document $x = ((N_x, E_x, r_x), \alpha)$ where
   • $N_x = \{r_x\} \bigcup_i N(X_i)$;
   • $E_x = \bigcup_i E(X_i) \cup \{(r_x, r_i, l_i) \mid r_i \in r(X_i)\}$;
   • r_x is a new node, not appearing in $N(X_i)$ for any i;
   • $\alpha = \bigcup_i \alpha(X_i)$.
– Otherwise, the conditions associated with the outgoing edges of the root are all unsatisfiable. Then $q^{\mathcal{D}}$ is the document $x = (t_x, \alpha)$ where $t_x = (\{r_x\}, \emptyset, r_x)$ and $\alpha(r_x) = \epsilon$ (the empty string).

Example 11. Figure 2 shows an SQL4X query that finds for each author the set of articles that she has written. The Tree-Datalog$_{4x}$ version of this query is also depicted in Figure 2, beside the query. Note that the condition associated with the edge labeled `bib` in the tree is an empty conjunction of atoms. Thus, there is one satisfying assignment of this edge's condition, the empty assignment.

5.2 Containment of Tree-Datalog$_{4x}$ Queries

In this section we consider the problem of determining containment of Tree-Datalog$_{4x}$ queries. As in Section 4.2, we use methods similar to those for standard Datalog queries.

Let $t = (N, E, r)$ and $t' = (N', E', r')$ be trees. A *tree homomorphism* from t to t' is a mapping μ from N to N', such that *(1)* the root is mapped to the root, i.e., $\mu(r) = r'$; and *(2)* edges are mapped to edges, i.e., $(n_1, n_2, l) \in E$ implies $(\mu(n_1), \mu(n_2), l) \in E'$. Let $x = (t, \alpha)$ and $x' = (t', \alpha')$ be documents. We say that x is *contained* in x' if there is a tree homomorphism μ from t to t' such that nonempty constants in the leaf nodes of x are preserved, i.e., for all leaf nodes n in t either $\alpha(n) = \alpha'(\mu(n))$ or $\alpha(n) = \epsilon$. We say that query q is *contained* in query q' if for all databases $\mathcal{D}$, it holds that $q^{\mathcal{D}}$ is contained in ${q'}^{\mathcal{D}}$.

Given a tree t we say that an edge e is *above* an edge e' if the path from the root of t to e' passes through e. If e is above e', we write $e < e'$. If e is either above e' or equal to e', we write $e \leq e'$.

Theorem 7. *Let $q = (t, \Psi, \tau)$ and $q' = (t', \Psi', \tau')$ be* Tree-Datalog$_{4x}$ *queries. Then $q \subseteq q'$ if there is a tree homomorphism μ from t to t' and a mapping θ of the variables in q' to the terms in q such that*

– *values in atomic nodes are equated, i.e., for all leaf nodes n in t*

$$\tau(n) = \theta(\tau'(\mu(n)));$$

– *for all edges e_1 in t and atoms a' in $\Psi'(\mu(e_1))$ there is an edge e_2 in t, with $e_2 \leq e_1$, such that*
  - *if a' is a relational atom or a label atom, then $\theta(a') \in \Psi(e_2)$* and
  - *if $a' = \langle u' \sim v' \rangle$ is a path atom, then either $\langle \theta(u') \sim \theta(v') \rangle \in \Psi(e_2)$ or $\langle \theta(v') \sim \theta(u') \rangle \in \Psi(e_2)$.*

For special cases it turns out that the condition in Theorem 7 is also a necessary condition. Let $q = (t, \Psi, \tau)$ be a query. Each path of edges $e_1, \ldots, e_m$ in t defines a Datalog$_{4x}$ query $q_{e_1,\ldots,e_m}$ as follows $q_{e_1,\ldots,e_m}() \leftarrow \Psi(e_1)$ & ... & $\Psi(e_m)$.

We say that a Tree-Datalog$_{4x}$ query $q = (t, \Psi, \tau)$ is *linear* if *(1)* t does not contain a node n with outgoing edges (n, n_1, l_1) and (n, n_2, l_2) such that $l_1 = l_2$; *and (2)* each path in t defines a linear Datalog$_{4x}$ query. We say that q is *exactly satisfiable* if every path in q, starting from q's root, defines an exactly satisfiable Datalog$_{4x}$ query. Exactly satisfiable linear queries are rather common. For example, the query in Figure 2 would be both exactly satisfiable and linear if self-path atoms were added for all node variables.

Theorem 8. *The condition in Theorem 7 is a necessary condition for containment of exactly satisfiable linear* Tree-Datalog$_{4x}$ *queries.*

Corollary 5. *Containment of exactly satisfiable linear* Tree-Datalog$_{4x}$ *queries can be decided in polynomial time.*

6 Related Work and Conclusion

Path queries are essentially queries defining path expressions. Path queries have been studied in [CDLV99,CDLV00,FLS98]. In [FLS98], containment of StruQL$_0$ queries was considered. Specifically, it was shown that testing containment of simple StruQL$_0$ queries is NP-complete. $\mathsf{Datalog}_{4x}$ queries without relational atoms can be written as a union of simple StruQL$_0$ queries. However, our results do not follow directly from the results in [FLS98] since *(1)* $\mathsf{Datalog}_{4x}$ is evaluated over a tree, while StruQL$_0$ is evaluated over a graph and *(2)* translating $\mathsf{Datalog}_{4x}$ queries into StruQL$_0$ incurs an exponential blowup.

There are two classical ways to answer a query against data sources that are both relational and XML. One option is to convert the XML into relations and then perform the query using a relational language, such as SQL. This is difficult because of the very varied structure of XML documents. The second option is to convert the relations to XML, and then use an XML query language. This method was explored in the SilkRoute system [FST00] which allows the definition of XML views over a relational database. In SQL4X we presented a third option. Our query language allows simultaneous querying of relations and XML and allows the user to choose the target data model. SQL4X also uses flexible semantics, which makes integrating varied documents with the same ontology easier (see Example 7).

XQuery [CCF+01] is one of the most prominent recently proposed query languages for XML. While SQL4X is appropriate for data integration, XQuery does not have the ability to query relations or to produce relations. In addition, SQL4X can be used as a *transformation language* to translate data between the relational and XML data models. SQL4X was defined as extension of SQL, and thus, can be efficiently implemented over a relational database system. XQuery, on the other hand, is a new language designed specifically for XML and probably will not be implemented in a relational database system. However, XQuery allows general XPath path expressions, whereas SQL4X currently has rather limited path expressions. This limitation allows SQL4X queries to be translated to SQL queries, for processing. SQL4X uses a flexible semantics, allowing the user to state that nodes are connected in some direction by a path. Flexible semantics can be captured in XQuery in a straight-forward fashion if the ancestor axis of XPath is supported. Support for this axis is currently under discussion since the ancestor axis does not appear in the abbreviated XPath syntax. Otherwise, translating an SQL4X query to XQuery may require an exponential blowup of the query.

Extending our characterizations for query containment to cases where queries contain functions from Table 1 is left for future work. We also intend to extend SQL4X to allow for use of general XPath expressions. The effect of such extensions on the complexity of containment is another important open problem.

References

AQM$^+$96. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. Technical report, The Stanford University Database Group, 1996.

ASU79. A.V. Aho, Y. Sagiv, and J.D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems*, 4(4):435–454, 1979.

CCF$^+$01. D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language, June 2001. Available at `http://www.w3.org/TR/xquery`.

CDLV99. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Rewriting of regular expressions and regular path queries. In Ch. Papadimitriou, editor, *Proc. 18th Symposium on Principles of Database Systems*, Philadelphia (Pennsylvania, USA), May 1999. ACM Press.

CDLV00. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Answering regular path queries using views. In *Proc. 16th International Conference on Data Engineering*, pages 389–398, San Diego (California, USA), March 2000. IEEE Computer Society.

CKPS95. S. Chaudhuri, S. Krishnamurthy, S. Potarnianos, and K. Shim. Optimizing queries with materialized views. In P.S.Yu and A.L.P. Chen, editors, *Proc. 11th International Conference on Data Engineering*, Taipei, March 1995. IEEE Computer Society.

Cla99. J. Clark. XSL transformations (XSLT specification), 1999. Available at `http://www.w3.org/ TR/WD-xslt`.

CM77. A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th Annual ACM Symposium on Theory of Computing*, 1977.

CNS99. S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In Ch. Papadimitriou, editor, *Proc. 18th Symposium on Principles of Database Systems*, Philadelphia (Pennsylvania, USA), May 1999. ACM Press.

CV93. S. Chaudhuri and M. Vardi. Optimization of real conjunctive queries. In *Proc. 12th Symposium on Principles of Database Systems*, Washington (D.C., USA), May 1993. ACM Press.

DFF$^+$98. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML, 1998. Available at `http://www.w3.org /TR/NOTE-xml-ql`.

DFF$^+$99. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *8th Int. World Wide Web Conference (WWW8)*, Toronto (Canada), May 1999. Available at `http://www8.org/fullpaper.html`.

DMD01. S. DeRose, E. Maler, and R. Daniel. XML pointer language (XPointer) 1.0, 2001. Available at `http://www.w3.org/TR/xptr`.

FLS98. D. Florescu, A.Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In J. Paredaens, editor, *Proc. 17th Symposium on Principles of Database Systems*, pages 139–148, Seattle (Washington, USA), June 1998. ACM Press.

FST00. M. Fernandez, D. Suciu, and W. Tan. Silkroute: Trading between relations and XML. In *Proc. 9th International World Wide Web Conference*, Amsterdam (Netherlands), May 2000. Available at `http://www9.org/w9cdrom`.

FSW99. M. Fernandez, J. Siméon, and P. Wadler. XML query languages: Experiences and exemplars, 1999. Available at `http://www.w3.org/1999/09/ql/docs/xquery.html`.

GL99. G. Grahne and L. V. S. Lakshmanan. On the difference between navigating semistructered data and querying it. In *8th International Workshop on Database Programming Languages*, Kinloch Rannoch (Scotland), September 1999.

GMW99. R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML/: Migrating the Lore data model and query language. In S. Cluet and T. Milo, editors, *Proc. 2nd International Workshop on the Web and Databases*, Philadelphia, (Pennsylvania, USA), June 1999.

IR95. Y.E. Ioannidis and R. Ramakrishnan. Beyond relations as sets. *ACM Transactions on Database Systems*, 20(3):288–324, 1995.

JK83. D.S. Johnson and A. Klug. Optimizing conjunctive queries that contain untyped variables. *SIAM Journal on Computing*, 12(4):616–640, 1983.

KS01. Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proc. 20th Symposium on Principles of Database Systems*, Santa Barbara (California, USA), May 2001. ACM Press.

Ley. M. Ley. DBLP website. Available at `http://www.informatik.uni-trier.de/~ley/db`.

LMSS93. A.Y. Levy, I. Singh Mumick, Y. Sagiv, and O. Shmueli. Equivalence, query-reachability, and satisfiability in Datalog extensions. In *Proc. 12th Symposium on Principles of Database Systems*, pages 109–122, Washington (D.C., USA), May 1993. ACM Press.

NSS98. W. Nutt, Y. Sagiv, and S. Shurin. Deciding equivalences among aggregate queries. In J. Paredaens, editor, *Proc. 17th Symposium on Principles of Database Systems*, pages 214–223, Seattle (Washington, USA), June 1998. ACM Press. Long version as Report of Esprit LTR DWQ.

RLS98. J. Robie, J. Lapp, and D. Schach. XML query language (XQL), 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.