Logic as a Query Language: from Frege to XML

Victor Vianu*

Univ. of California at San Diego, CSE 0114, La Jolla, CA 92093-0114, USA

Abstract. This paper surveys some of the more striking aspects of the connection between logic and databases, from the use of first-order logic as a query language in classical relational databases all the way to the recent use of logic and automata theory to model and reason about XML query languages. It is argued that logic has proven to be a consistently effective formal companion to the database area.

1 Introduction

The database area is concerned with storing, querying and updating large amounts of data. Logic and databases have been intimately connected since the birth of database systems in the early 1970's. Their relationship is an unqualified success story: indeed, first-order logic (FO), which can be traced all the way to Frege [15], lies at the core of modern database systems. Indeed, the standard query languages such as *Structured Query Language* (SQL) and *Query-By-Example* (QBE) are syntactic variants of FO. More powerful query languages are based on extensions of FO with recursion, and are reminiscent of the well-known fixpoint queries studied in finite-model theory. Lately, logic and automata theory have proven to be powerful tools in modeling and reasoning about the Extended Markup Language (XML), the current *lingua franca* of the Web. The impact of logic on databases is one of the most striking examples of the effectiveness of logic in computer science.

In this paper we survey some aspects of the connection between logic and databases. In Section 2 we discuss the use of FO as a query language in relational databases. We then look briefly at query languages beyond FO and point out the connection between the theory of query languages and finite model theory. In Section 3 we discuss how logic and automata theory provide the formal foundation for XML query languages and their static analysis.

A good introduction to the database area may be found in [40], while [42] provides a more in-depth presentation. The first text on database theory is [24], followed more recently by [2]. The latter text also described database query languages beyond FO, including Datalog and fixpoint logics. An excellent survey of early relational database theory is provided in [23].

* Work supported in part by the National Science Foundation under grant number IIS-9802288.

2 FO as a Query Language

This section[2] discusses the following question: Why has FO turned out to be so successful as a query language? We will focus on three main reasons:

- FO has syntactic variants that are easy to use. These are used as basic building blocks in practical languages like SQL and QBE.
- FO can be efficiently implemented using *relational algebra*, which provides a set of simple operations on relations expressing all FO queries. Relational algebra as used in the context of databases was introduced by Ted Codd in [12]. It is related to Tarski's Cylindric Algebras [18]. The algebra turns out to yield a crucial advantage when large amounts of data are concerned. Indeed, the realization by Codd that the algebra can be used to efficiently implement FO queries gave the initial impetus to the birth of relational database systems[3].
- FO queries have the potential for "perfect scaling" to large databases: if massive parallelism is available, FO queries can *in principle* be evaluated in *constant time*, independent of the database size.

A relational database can be viewed as a finite relational structure. Its signature is much like a relational FO signature, with the minor difference that relations and their coordinates have names. The name of a coordinate is called an *attribute*, and the set of attributes of a relation R is denoted $att(R)$. For example, a "beer drinker's" database might consist of the following relations:

frequents	*drinker*	*bar*
	Joe	*King's*
	Joe	*Molly's*
	Sue	*Molly's*
	...	...

serves	*bar*	*beer*
	King's	*Bass*
	King's	*Bud*
	Molly's	*Bass*
	...	...

The main use of a database is to query its data: Which passengers are connecting from flight 173 to flight 645? List all students who take calculus and orchestra. Find the drinkers who frequent only bars serving Bass. It turns out that each query expressible in FO can be broken down into a sequence of very simple subqueries. Each subquery produces an intermediate result, which may be used by subsequent subqueries. A subquery is of the form:

$$R \;:=\; \exists \mathbf{x}\ (L_1 \wedge \ldots \wedge L_k),$$

where L_i is a literal $P(\mathbf{y})$ or $\neg P(\mathbf{y})$, P is in the input signature or is on the left-hand side of a previous subquery in the sequence, and R is not in the input signature and does not occur previously in the sequence. The meaning of such a subquery is to assign to R the result of the FO query $\exists \mathbf{x}\ (L_1 \wedge \ldots \wedge L_k)$

[2] This section is based on portions of [17].

[3] Codd received the Turing Award for his work leading to the development of relational systems.

on the structure resulting from the evaluation of the previous subqueries in the sequence. The subqueries provide appealing building blocks for FO queries. This is illustrated by the language QBE, in which a query is formulated as just described. For example, consider the following query on the "beer drinker's" database:

Find the drinkers who frequent some bar serving Bass.

This can be expressed by a single query of the above form:

$$(\dagger)\ \ answer \ := \ \exists b\ (frequents(d,b) \wedge serves(d, Bass)).$$

In QBE, the query is formulated in a visually appealing way as follows:

$$\begin{array}{c|c} answer & drinker \\ \hline & d \end{array} \leftarrow \begin{array}{c|cc} frequents & drinker & bar \\ \hline & d & b \end{array} \ \begin{array}{c|cc} serves & bar & beer \\ \hline & b & Bass \end{array}$$

Similar building blocks are used in SQL, the standard query language for relational database systems.

Let us consider again the query ($\dagger$). The naive implementation would have us check, for each drinker d and bar b, whether $frequents(d,b) \wedge serves(d, Bass)$ holds. The number of checks is then the product of the number of drinkers and the number of bars in the database, which can be roughly n^2 in the size of the database. This turns out to be infeasible for very large databases. A better approach, and the one used in practice, makes use of relational algebra. Before discussing how this works, we informally review the algebra's operators. There are two set operators, $\cup$ (union) and $-$ (difference). The *selection* operator, denoted $\sigma_{cond}(R)$ extracts from R the tuples satisfying a condition *cond* involving (in)equalities of attribute values and constants. For example, $\sigma_{beer=Bass}(serves)$ produces the tuples in *serves* for which the beer is *Bass*. The *projection* operator, denoted $\pi_X(R)$, projects the tuples of relation R on a subset X of its attributes. The *join* operator, denoted by $R \bowtie Q$, consists of all tuples t over $att(R) \cup att(Q)$ such that $\pi_{att(R)}(t) \in R$ and $\pi_{att(Q)}(t) \in Q$. A last unary operator allows to *rename* an attribute of a relation without changing its contents.

Expressions constructed using relational algebra operators are called relational algebra queries. The query ($\dagger$) is expressed using relational algebra as follows:

$$(\ddagger)\ \ \pi_{drinker}(\sigma_{beer=Bass}(frequents \bowtie serves)).$$

A result of crucial importance is that *FO and relational algebra express precisely the same queries.*

The key to the efficient implementation of relational algebra queries is twofold. First, individual algebra operations can be efficiently implemented using data structures called *indexes*, providing fast access to data. A simple example of such a structure is a binary search tree, which allows locating the tuples with a given attribute value in time $log(n)$, where n is the number of tuples. Second, algebra queries can be simplified using a set of *rewrite rules* applied by the query processor. The query (‡) above can be rewritten in the equivalent but more efficient form:

$$\pi_{drinker}[frequents \bowtie \pi_{bar}(\sigma_{beer=Bass}(serves))].$$

The use of indexes and rewriting rules allows to evaluate the above query at cost roughly $n\ log(n)$ in the size of the database, which is much better than n^2. Indeed, for very large databases this can make the difference between infeasibility and feasibility.

The FO queries turn out to be extremely well-behaved with respect to *scaling*: given sufficient resources, response time can *in principle* be kept constant as the database becomes larger. They key to this remarkable property is parallel processing. Admittedly, *a lot* of processors are needed to achieve this ideal behavior : polynomial in the size of the database. This is unlikely to be feasible in practice any time soon. The key point, however, is that FO query evaluation admits *linear* scaling; the speed-up is proportional to the number of parallel processors used.

Once again, relational algebra plays a crucial role in the parallel implementation of FO. Indeed, the algebra operations are *set oriented*, and thus highlight the intrinsic parallelism in FO queries. For example, consider the projection $\pi_X(R)$. The key observation is that one can project the tuples in R independently of each other. Given one processor for each tuple in R, the projection can be computed in constant time, independent of the number of tuples. As a second example, consider the join $R \bowtie Q$. This can be computed by joining all *pairs* of tuples from R and Q, independently of each other. Thus, if one processor is available for each pair, the join can be computed in constant time, independent on the number of tuples in R and Q.

Since each algebra operation can be evaluated in constant parallel time, each algebra query can also be evaluated in constant time. The constant depends only on the query, and is independent of the size of the database. Of course, more and more processors are needed as the database grows.

In practice, the massive parallelism required to achieve perfect scaling is not available. Nevertheless, there are algorithms that can take optimal advantage of a given set of processors. It is also worth noting that the processors implementing the algebra need not be powerful, as they are only required to perform very specific, simple operations on tuples. In fact, it is sufficient to have processors that can implement the basic Boolean circuit operations. This fact is formalized by a result due to Immerman [21] stating that FO is included in AC_0, the class of problems solvable by circuits of constant depth and polynomial size, with unbounded fan-in.

Beyond FO The inability of FO to express certain useful queries, such as connectivity of finite graphs, was noted early on by database theoreticians ([14], and independently [4]). Such facts have led to the introduction of a variety of database languages extending FO with recursion. The proposed paradigms are quite diverse and include SQL extensions, fixpoint logics, imperative programming, logic programming, and production systems (e.g., see [2]).

The wide diversity of query language paradigms beyond FO may be disconcerting at first. It may appear that there is no hope for unity. However, despite the wide variations, many of the languages are equivalent. In fact, two central classes of queries emerge, known as the *fixpoint* queries and the *while* queries. The *fixpoint* queries are equivalent to the well-known fixpoint logics FO+IFP and FO+LFP, and the *while* queries are equivalent to FO+PFP. Recall that FO+LFP extends FO with an operator LFP that iterates a positive formula up to a fixpoint; FO+IFP extends FO with an operator IFP that iterates an arbitrary FO formula up to a fixpoint. Convergence is ensured by forcing the iteration to be cumulative (whence the name of inflationary fixpoint operator). And, FO+PFP extends FO with an operator PFP that iterates an arbitrary FO formula, and may or may not reach a fixpoint (whence the name of partial fixpoint operator). The connection of the database languages to the logics FO+IFP (FO+LFP) and FO+PFP underscores the intimate connection between practical query languages and logics studied in finite-model theory. The relationship between finite-model theory and databases is discussed in more detail in [44].

3 XML

Much of the recent research in databases has been motivated by the Web scenario. As the standard for data exchange on the Web, the Extended Markup Language (XML) plays a central role in current databases (see [1]). As it turns out, logic, particularly automata theory, plays a crucial role in modeling and reasoning about XML query languages. As an aside, we note that automata theory has long interacted with database theory (see [43] for a survey).

An XML document consists of nested elements, with ordered sub-elements. Each element has a name (also called tag or label). The full XML has many bells and whistles, but its simplest abstraction is as a labeled ordered tree (with labels on nodes), possibly with data values associated to the leaves. For example, an XML document holding ads for used cars and new cars is shown in Figure 1 (left), together with its abstraction as a labeled tree (right, data values omitted).

DTDs and XML Schemas The basic typing mechanism for XML documents is called *Document Type Definition (DTD)*. Let Σ be a finite alphabet of labels. A DTD consists of a set of rules of the form $e \to r$ where $e \in \Sigma$ and r is a regular expression over Σ. There is one such rule for each e, and the DTD also specifies the label of the root. An XML document satisfies a DTD if, for each rule $e \to r$, and each node labeled e, the sequence of labels of its children spells a word in r. For example, a DTD might consist of the rules (with ϵ-rules omitted):

$root : section;$

$section \to intro, section^*, conc$

An example of a labeled tree satisfying the above DTD is:

```
<dealer>
 <UsedCars>
  <ad>
   <model>Honda</model>
   <year>92</year>
  </ad>
 </UsedCars>
 <NewCars>
  <ad>
   <model>BMW</model>
  </ad>
 </NewCars>
</dealer>
```

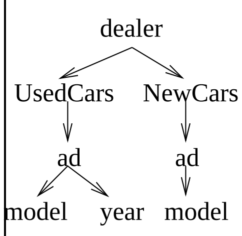
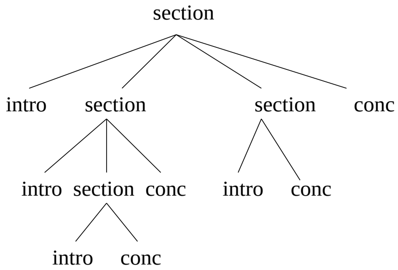


Fig. 1. Dealer XML document (data values omitted from tree representation)



Thus, each DTD d defines a set of labeled ordered trees, denoted by $sat(d)$. It turns out that basic DTDs have many limitations as schema languages. Some are addressed in extensions that have been proposed, such as XML Schema, that are still in a state of flux. One important limitation of DTDs is the inability to separate the *type* of an element from its *name*. For example, consider the dealer document in Figure 1. It may be natural for used car ads to have different structure than new car ads. There is no mechanism to do this using DTDs, since rules depend only on the name of the element, and not on its context. To overcome this limitation, XML Schema provides a mechanism to decouple element names from their types and thus allow context-dependent definitions of their structure. One way to formalize the decoupling of names from types is by the notion of *specialized DTD*, studied in [37] and equivalent to formalisms proposed in [6, 11]. Specialization is present in the XQuery typing system and also (in a restricted form) in XML-Schema. The idea is to use whenever necessary "specializations" of element names with their own type definition. For example, this allows us to use in the DTD two variants of ads with different structure: used car ads, and new car ads.

Many basic questions arise in connection to schemas for XML. How hard is it to check validity of an XML document with respect to a schema? When can a set of XML documents be characterized by a schema? Is there always a most precise schema describing a given set of XML documents? Can the union, difference, intersection of sets of valid documents specified by schemas be in turn described by another schema? If yes, how can that schema be computed? It turns out that there is a unified approach to such questions based on a powerful connection between DTDs and tree automata, discussed next. Indeed, specialized DTDs define precisely the *regular tree languages*.

DTDs and tree automata We informally review the notion of regular tree language and tree automaton. Tree automata are devices whose function is to accept or reject their input, which in the classical framework is a complete binary tree with nodes labeled with symbols from some finite alphabet Σ. There are several equivalent variations of tree automata. A non-deterministic top-down tree automaton over Σ has a finite set Q of states, including a distinguished initial state q_0 and an accepting state q_f. In a computation, the automaton labels the nodes of the tree with states, according to a set of rules, called *transitions*. An internal node transition is of the form $(a, q) \to (q', q'')$, for $a \in \Sigma$. It says that, if an internal node has symbol a and is labeled by state q, then its left and right children may be labeled by q' and q'', respectively. A leaf transition is of the form $(a, q) \to q_f$ for $a \in \Sigma$. It allows changing the label of a leaf with symbol a from q to the accepting state q_f. Each computation starts by labeling the root with the start state q_0, and proceeds by labeling the nodes of the trees non-deterministically according to the transitions. The input tree is accepted if *some* computation results in labeling all leaves by q_f. A set of complete binary trees is *regular* iff it is accepted by some top-down tree automaton. Regular languages of finite binary trees are surveyed in [16].

There is a *prima facie* mismatch between DTDs and tree automata: DTDs describe unranked trees, whereas classical automata describe binary trees. There are two ways around this. First, unranked trees can be encoded in a standard way as binary trees. Alternatively, the machinery and results developed for regular tree languages can be extended to the unranked case, as described in [8] (an extension for unranked infinite trees is described in [3]). Either way, one can prove a surprising and satisfying connection between specialized DTDs and tree automata: *they are precisely equivalent* [8, 37].

The equivalence of specialized DTDs and tree automata is a powerful tool for understanding XML schema languages. Properties of regular tree languages transfer to specialized DTDs, including closure under union, difference, complement, decidability of emptiness (in PTIME) and inclusion (in EXPTIME), etc. Moreover, automata techniques can yield algorithmic insight into processing DTDs. For example, the naive algorithm for checking validity of an XML document with respect to a specialized DTD is exponential in the size of the document (due to guessing specializations for labels). However, the existence of a bottom-up deterministic automaton equivalent to the specialized DTD shows that validity can be checked in linear time by a single bottom-up pass on the document.

XML query languages and tree transducers There has been a flurry of proposals for XML query languages, including XML-QL [13], XSLT (W3C Web site), XQL

[38], XDuce [19, 20], and Quilt [10]. Recently, the language XQuery has been adopted by the W3C committee as the standard query language for XML.

XML query languages take trees as input and produce trees as output. Despite their diversity, it turns out that their tree manipulation capabilities are subsumed by a single model of tree transducer, called *k-pebble transducer* [26]. This provides a uniform framework for measuring the expressiveness of XML languages, and it is instrumental in developing static analysis techniques. In particular, as we shall see, the transducers can be used for typechecking XML queries. The k-pebble transducer uses up to k pebbles to mark certain nodes in the tree. Transitions are determined by the current node symbol, the current state, and by the existence/absence of the various pebbles on the node. The pebbles are ordered and numbered $1, 2, \ldots, k$. The machine can place pebbles on the root, move them around, and remove them. In order to limit the power of the transducer the use of pebbles is restricted by a stack discipline: pebbles are placed on the tree in order and removed in reverse order, and only the highest-numbered pebble present on the tree can be moved. The transducer works as follows. The computation starts by placing pebble 1 on the root. At each point, pebbles $1, 2, \ldots, i$ are on the tree, for some $i \in \{1, \ldots, k\}$; pebble i is called the *current pebble*, and the node on which it sits is the *current node*. The current pebble serves as the head of the machine. The machine decides which *transition* to make, based on the following information: the current state, the symbol under the current pebble, and the presence/absence of the other $i - 1$ pebbles on the current node. There are two kinds of transitions: *move* and *output* transitions. *Move* transitions are of four kinds: they can place a new pebble, pick the current pebble, or move the current pebble in one of the four directions *down-left, down-right, up-left, up-right* (one edge only). If a move in the specified direction is not possible, the transition does not apply. After each move transition the machine enters a new state, as specified by the transition. An *output* transition emits some labeled node and does not move the input head. There are two kinds of output transitions. In a *binary* output the machine spawns two computation branches computing the left and right child respectively. Both branches inherit the positions of all pebbles on the input, and do not communicate; each moves the k pebbles independently of the other. In a *nullary* output the node being output is a leaf and that branch of computation halts.

Looking at the global picture, the machine starts with a single computation branch and no output nodes. After a while it has constructed some top fragment of the output tree, and several computation branches continue to compute the remaining output subtrees. The entire computation terminates when all computation branches terminate. It turns out that all transformations over unranked trees over a given finite alphabet expressed in existing XML query languages (XQuery, XML-QL, Lorel, StruQL, UnQL, and XSLT) can be expressed as k-pebble transducers. This does *not* extend to queries with joins on data values, since these require an infinite alphabet. Details, as well as examples, can be found in [26].

Typechecking using tree automata and transducers Typechecking by means of tree automata and k-pebble transducers is explored in [26]. As discussed above, this applies to the tree manipulation core of most XML languages. Typechecking can be done by means of *inverse type inference*. Suppose d is an input specialized DTD (or, equivalently, a tree automaton), and d' an output specialized DTD. Consider a k-pebble transducer T. It can be shown that $T^{-1}(sat(d'))$ is always a regular tree language, for which a tree automaton can be effectively constructed from T and d'. Then typechecking amounts to checking that $sat(d) \subseteq T^{-1}(sat(d'))$, which is decidable.

There are several limitations to the above approach. First, the complexity of typechecking in its full generality is very high: a tower of exponentials of height equal to the number of pebbles, so non-elementary. Thus, general typechecking appears to be prohibitively expensive. However, the approach can be used in restricted cases of practical interest for which typechecking can be reduced to emptiness of automata with *very few pebbles*. Even one or two pebbles can be quite powerful. For example, typechecking selection XML-QL queries without joins (i.e., queries that extract the list of bindings of a variable occurring in a tree pattern) can be reduced to emptiness of a 1-pebble automaton with exponentially many states. Another limitation has to do with data values. In general, the presence of data values leads to undecidability of typechecking. For example, if k-pebble transducers are extended with equality tests on the data values sitting under the pebbles, even emptiness is undecidable. However, the approach can be extended to restricted classes of queries with data value joins [5]. An overview of typechecking for XML is provided in [41].

Other models Another transducer model for XML queries, called *query automaton*, is described in [35]. This work was the first to use MSO to study query languages for XML. Query automata, however, differ significantly from k-pebble transducers: they take an XML input tree and return a set of nodes in the tree. By contrast a k-pebble transducer returns a new output tree. Several abstractions of XML languages are studied in [25], and connections to extended tree-walking transducers with look-ahead are established. Various static analysis problems are considered, such as termination, emptiness, and usefulness of rules. It is also shown that ranges of the transducers are closed under intersection with *generalized DTDs* (defined by tree regular grammars). Tree-walking automata and their relationship to logic and regular tree languages are further studied in [34].

Another computation model for trees, based on *attribute grammars*, is considered in [32]. These capture queries that return sets or tuples of nodes from the input trees. Two main variants are considered. The first expresses all unary queries definable by MSO formulas. The second captures precisely the queries definable by first-order inductions of linear depth. Equivalently, these are the queries computable on a parallel random access machine with polynomially many processors. These precise characterizations in terms of logic and complexity suggest that attribute grammars provide a natural and robust querying mechanism for labeled trees.

To remedy the low expressiveness of pattern languages based on regular path expressions, a guarded fragment of MSO that is equivalent to MSO but that can be evaluated much more efficiently is studied in [33, 39]. For example, it is shown that this fragment of MSO can express FO extended with regular path expressions. In [7] a formal model for XSLT is defined incorporating features like modes, variables, and parameter passing. Although this model is not computational complete, it can simulate k-pebble transducers, even extended with equality tests on data values. Consequently, and contrary to conventional wisdom, XSLT can simulate all of XML-QL!

Feedback into automata theory The match between XML and automata theory is very promising, but is not without its problems. The classical formalism sometimes needs to be adapted or extended to fit the needs of XML. For example, tree automata are defined for ranked trees, but XML documents are unranked trees. This required extending the theory of regular tree languages to unranked trees [8], and has given rise to a fertile line of research into formalisms for unranked trees. This includes extensions of tree transducers [25], push-down tree automata [27], attribute grammars [28], and caterpillar expressions [9]. Another mismatch arises from the fact that XML documents have data values, corresponding to trees over *infinite* alphabets. Regular tree languages over infinite alphabets have not been studied, although some investigations consider the string case [22, 36]. Tree-walking transducers accessing data values of XML documents are considered in [31]. Informative surveys on logic and automata-theoretic approaches to XML are provided in [30, 29].

4 Conclusion

We first discussed why logic has proven to be a spectacularly effective tool in classical relational databases. FO provides the basis for the standard query languages, because of its ease of use and efficient implementation via relational algebra. FO can achieve linear scaling, given parallel processing resources. Thus, its full potential as a query language remains yet to be realized.

Beyond FO, there is a wide variety of database languages extending standard query languages with recursion. Despite the seemingly hopeless diversity, logic unifies the landscape. Indeed, most languages converge around two central classes of queries: the *fixpoint* queries, defined by the logics FO+LFP and FO+IFP, and the *while* queries, defined by FO+PFP. This underscores an intimate connection between the theory of query languages and finite-model theory, that has led to much fruitful cross-fertilization.

Recently, a lot of research has focused on XML, the new standard for data exchange on the Web. Again, logic, particularly automata theory, has provided crucial tools for modeling and reasoning about XML query languages.

Altogether, the impact of logic on databases is one of the most striking examples of the effectiveness of logic in computer science.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan Kauffman, 1999.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
3. S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. *JACM*, 45(5):798–842, 1998. Extended abstract in SIGMOD'89.
4. A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 110–117, 1979.
5. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: typechecking revisited. In *Proc. ACM PODS*, 2001.
6. C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Int'l. Conf. on Database Theory*, pages 296–313, 1999.
7. G. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. In *Proc. DOOD*, pages 1137–1151, 2000.
8. A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets, 2001. Technical Report HKUST-TCSC-2001-0, Hong-Kong University of Science and Technology.
9. A. Brüggemann-Klein and D. Wood. Caterpillars: a context specification technique. *Markup Languages*, 2(1):81–106, 2000.
10. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, pages 53–62, 2000.
11. S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. ACM SIGMOD Conf.*, pages 177–188, 1998.
12. E. F. Codd. A relational model for large shared databank. *Communications of the ACM*, 13(6):377–387, June 1970.
13. A. Deutsch, M. Fernandez, D.Florescu, A. Levy, and D. Suciu. A query language for XML. In *WWW8*, pages 11–16, 1999.
14. R. Fagin. Monadic generalized spectra. *Z. Math. Logik*, 21:89–96, 1975.
15. G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* L. Nebert, Halle a. S., 1879.
16. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1, pages 1–68. Springer, 1997.
17. J. Halpern, R. Harper, N. Immerman, P. Kolaitis, M. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001.
18. L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras: Part I.* North Holland, 1971.
19. H. Hosoya and B. Pierce. XDuce: A typed XML processing language (Preliminary Report). In *WedDB (Informal Proceedings)*, pages 111–116, 2000.
20. H. Hosoya, J. Vouillon, and B. Pierce. Regular expression types for XML. In *Int. Conf. on Functional Programming*, pages 11–22, 2000.
21. N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.
22. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
23. P. Kannelakis. Elements of relational database theory. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1074–1156. Elsevier, 1991.

24. D. Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1983.
25. S. Maneth and F. Neven. Structured document transformations based on XSL. In *Proc. DBPL*, pages 79–96. LNCS, Springer, 1999.
26. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proc. ACM PODS*, pages 11–22, 2000. Full paper to appear in special issue of JCSS.
27. A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, pages 134–145. LNCS, Springer, 1998.
28. F. Neven. Extensions of attribute grammars for structured document queries. In *Proc. DBPL*, pages 97–114. LNCS, Springer, 2000.
29. F. Neven. Automata, logic, and XML. In *Proc. Computer Science Logic*, pages 2–26. Springer LNCS, 2002.
30. F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.
31. F. Neven. On the power of walking for querying tree-structured data. In *Proc. ACM PODS*, pages 77–84, 2002.
32. F. Neven and J. V. den Bussche. Expressiveness of structured document query languages based on attribute grammars. *JACM*, 49(1), 2002. Extended abstract in PODS 1998.
33. F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. ACM PODS*, pages 145–156, 2000.
34. F. Neven and T. Schwentick. On the power of tree-walking automata. In *Proc. ICALP*, pages 547–560, 2000.
35. F. Neven and T. Schwentick. Query automata on finite trees. *Theoretical Computer Science*, 275(1-2):633–674, 2002.
36. F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. In *Proc. MFCS*, pages 560–572, 2001.
37. Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. ACM PODS*, pages 35–46, 2000.
38. J. Robbie, J. Lapp, and D. Schach. XML query language (XQL). In *The Query Languages Workshop (QL'98)*, 1998.
39. T. Schwentick. On diving in trees. In *Proc. MFCS*, pages 660–669, 2000.
40. A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts.* Mc Graw Hill, 1997.
41. D. Suciu. The XML typechecking problem. *SIGMOD Record*, 31(1):89–96, 2002.
42. J. D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, Rockville, MD 20850, 1989.
43. M. Vardi. Automata theory for database theoreticians. In *Proc. ACM PODS*, pages 83–92, 1989.
44. V. Vianu. Databases and finite-model theory. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 31, 1997.

This article was processed using the LaTeX macro package with LLNCS style