A Web Odyssey: from Codd to XML

Victor Vianu[*†]
U.C. San Diego
vianu@cs.ucsd.edu

1 Introduction

The Web presents the database area with vast opportunities and commensurate challenges. Databases and the Web are organically connected at many levels. Web sites are increasingly powered by databases. Collections of linked Web pages distributed across the Internet are themselves tempting targets for a database. The emergence of XML as the *lingua franca* of the Web brings some much needed order and will greatly facilitate the use of database techniques to manage Web information.

This paper discusses some of the developments related to the Web from the viewpoint of database theory. As we shall see, the Web scenario requires revisiting some of the basic assumptions of the area. To be sure, database theory remains as valid as ever in the classical setting, and the database industry will continue to represent a multi-billion dollar target of applicability for the foreseeable future. But the Web represents an opportunity of an entirely different scale. We are thus at an important juncture. Database theory could retain its classical focus and turn inwards. Or, it could attempt to take heads-on the challenge of the Web and contribute to an important part of its formal foundations. To do so, it will have to leave its familiar shores and reinvent itself. There are good signs that the journey has already begun.

What makes the Web scenario different from classical databases? In short, everything. A classical database is a coherently designed system. The system imposes rigid structure, and provides queries, updates, as well as transactions, concurrency, integrity, and recovery, in a controlled environment. The Web escapes any such control. It is a free-evolving, ever-changing collection of data sources of various shapes and forms, interacting according to a flexible protocol. A database is a polished artifact. The Web is closer to a natural ecosystem.

Why bother then? Because there is tremendous need for database-like functionality to efficiently provide and access data on the Web and for a wide range of applications. And, despite the differences, it turns out that database know-how remains extremely valuable and effective. The design of XML query and schema languages has been heavily influenced by the database community. XML query processing techniques are based on underlying algebras, and use rewrite rules and execution plans much like their relational counterparts. The use of the database paradigm on the Web is a success story, a testament to the robustness of databases as a field.

Much of the traditional framework of database theory needs to be reinvented in the Web scenario. Data no longer fits nicely into tables. Instead, it is self-describing and irregular, with little distinction between schema and data. This has been formalized by *semi-structured* data. Schemas, when available, are a far cry from tables, or even from more complex object-oriented schemas. They provide much richer mechanisms for specifying flexible, recursively nested structures, possibly ordered. A related problem is that of *constraints*, generalizing to the semi-structured and XML frameworks classical dependencies like functional and inclusion dependencies. Specifying them often requires recursive navigation through the nested data, using path expressions.

Query languages also differ significantly from their relational brethren. The lack of schema leads to a more navigational approach, where data is explored from specific entry points. The nested structure of data leads to recursion in queries, in the form of path expressions. Other paradigms have also proven useful, such as structural recursion. Query languages typically provide mechanisms to construct complex answers. The resulting classes of queries are not always neat (for example some query languages are not even closed under composition) so their expressiveness is not easy to characterize. The complexity of queries is also hard to evaluate in a relevant way by traditional means (can a query of complexity LOGSPACE in the size of the Web be called tractable?). As a corollary to the rich schema formalisms, query *typechecking* has become an important issue.

The development of Internet technology has occurred very rapidly, initially leaving theory behind. As is often the case in such situations, practical development sometimes seemed more ad-hoc than well principled. But, as has also happened before, order and formal beauty have nonetheless emerged in surprising and satisfying ways. One of the most elegant theoretical developments is the connection of XML schemas and queries to tree automata. Indeed, while the classical theory of queries languages is intimately related to

[*]This is an abridged version of the paper with the same title from PODS 2001. Work supported in part by the National Science Foundation under grant number IIS-9802288.

[†]Database Principles Column. Column editor: Leonid Libkin, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3H5, Canada. E-mail: libkin@cs.toronto.edu.

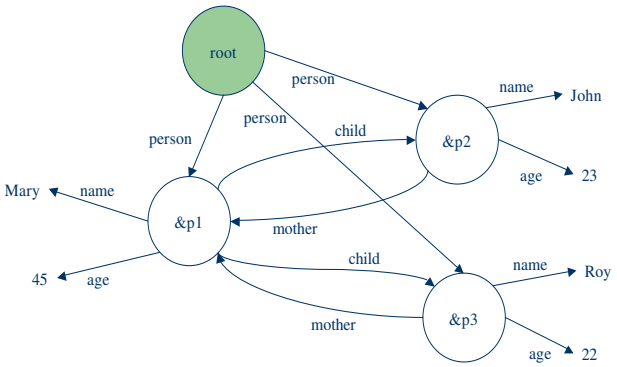


Figure 1: A semi-structured data graph

finite-model theory, automata theory has instead emerged as the natural formal companion to XML. Interestingly, research on XML is feeding back into tree automata theory and is re-energizing this somewhat arcane area of language theory. This connection is a recurring theme throughout the paper.

In the next section we discuss data on the Web, including semi-structured data and XML, schemas, and constraints. Section 3 deals with queries on the Web. Section 4 discusses query typechecking.

This paper is by no means a comprehensive survey of the developments in database theory related to the Web. Several excellent articles serving this purpose are referenced in the paper. The book [2] is an invaluable source of information on databases and the Web.

2 Data, Schemas, Constraints

To begin, we discuss the kinds of data found on the Web, and mechanisms to describe its structure by schemas and constraints.

2.1 Data on the Web

The Web is a fascinating target for databases. But viewing the Web as one huge database to be queried is a daunting proposition. Data on the Web is irregular, heterogeneous, and globally distributed. The lack of common structure and meaning makes it difficult to locate data relevant to a query or to relate information from different sources. But there is worse: the Web as a whole is in some sense a fictional, virtual object. Like the blind men discovering the elephant, centralized repositories of Web data can only retrieve by crawling small, locally consistent fragments of the Web, many of which rapidly become stale. There seems to be an *uncertainty principle* at work: it is not possible to capture, let alone maintain, a consistent snapshot of the entire Web. This is radically different from the database framework, where queries have full access to their input. Short of a well-defined input, the very meaning of querying the Web is open to discussion.

There can be no single, well-defined notion of the data that can be found on the Web. One must focus on specific aspects and levels of the Web to find the data that best fits one's needs. One might be interested in the graph of all Web pages and their hyperlinks. Or, one might wish to focus on XML documents available on the Web and their internal structure. Alternatively, one might be interested

R	A	B	C
	1	1	2
	2	1	3

Q	C	D
	2	1
	1	0

Figure 2: A relational database

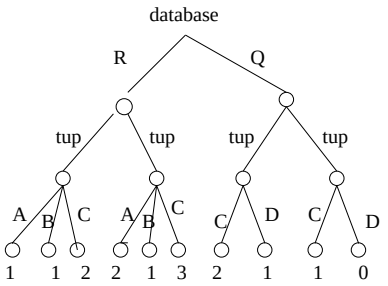


Figure 3: Data graph for R and Q

to provide or retrieve information from databases exporting XML views or interacting with the outside world through forms providing limited access.

2.2 Semi-structured data and XML

Semi-structured data is a bare-bones abstraction of the irregular, self-describing data found across the Web. It is also motivated by applications such as scientific databases, and the integration of heterogeneous data.

Semi-structured data is a labeled graph. The nodes are viewed as objects and have object ids. They can be atomic or complex. Complex objects are linked to other objects by labeled edges. Atomic objects have data values associated with them. The intent is that schema and data be represented in the same way, and this yields a very flexible and powerful formalism for describing data in a unified manner. Figure 1 shows one such data graph. Relational or object-oriented databases can also be represented as graphs. For example, the database in Figure 2 is represented by the data graph in Figure 3. Note that there is no explicit distinction between data and schema in the graph.

Several variants of the semi-structured data model have been proposed, with minor differences in formalism. The

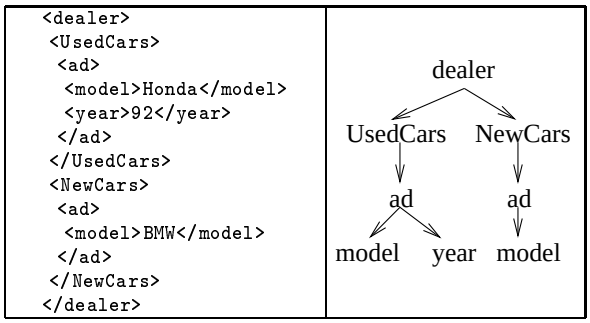


Figure 4: Dealer XML document

first semi-structured data model was the Object Exchange Model (OEM), introduced within the Tsimmis project as a vehicle for integrating heterogeneous sources [40, 25, 64]. This was soon followed by Lore [4, 50]. Another model, UnQL, was developed at the University of Pennsylvania [67, 19], motivated by the OEM model and by the ACeDB graph model used in biological databases [72].

Unlike the semi-structured data models, XML (Extended Markup Language) does not originate in the database community. It has been introduced in the document community as a subset of SGML. XML is in some sense an augmentation of HTML allowing annotating data with information about its *meaning* rather than just its presentation. 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 4 (left), together with its abstraction as a labeled tree (right, data values omitted).

The emergence of XML has placed increased importance on labeled trees capturing the structure of XML documents. However, XML additionally provides a referencing mechanism among elements that allows simulating arbitrary graphs, and so, semi-structured data. This aspect has been left out of some formal models for XML, because neither XML schemas nor query languages take advantage of it.

It is worth mentioning that XML can also be viewed as an object model. This is illustrated by a standard API for XML proposed by W3C, where XML documents are described in terms of the *Document Object Model* (DOM). Other extensions to XML and DTDs proposed by W3C, such as RDF, also have an object-oriented flavor (see [2] and the W3C Web site).

2.3 Schemas

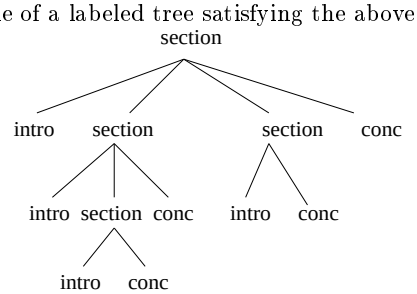
Schemas for semi-structured data. The flexibility of semi-structured data comes at a price: the loss of schema. But schemas are very useful. They describe the data and help query it, and allow query optimization and efficient storage. To retain some of these advantages, there have been attempts to recover schema information from semi-structured data. This has led to proposals such as *data guides* [38] and *representative objects* [53]. More discussion of schemas for semi-structured data can be found in [73].

Schemas for XML. XML marks the "return of the schema" in semistructured data, in the form of its Data Type Definitions (DTDs). More recently, many schema languages extending DTDs have been proposed, including XML-Schema, DSD, SOX, RELAX, etc. Most proposals can be found as technical reports of the W3C (http://www.w3c.org). Comparative presentations of several XML schema languages can be found in [2] and [47]. A survey of the XML schema languages is beyond the scope of this paper. We will focus here on DTDs and a very useful extension allowing to decouple the definition of the structure of an element from its name.

Essentially, a DTD is an extended context-free grammar. The non-terminals of the grammar are the labels (tags) of elements in the labeled tree corresponding to the XML document. There are no terminal symbols. 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 it is a derivation of the extended context-free grammar. That is, for label e with associated 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:



Thus, each DTD d defines a set of labeled ordered trees, denoted by $sat(d)$.

It turns out that DTDs have many limitations as schema languages. Some are addressed in the many extensions that have been proposed, which are still in a state of flux. For example, the set of tree languages definable by DTDs is not even closed under union (nor other Boolean operators). Another important limitation is the inability to separate the *type* of an element from its *name*. For example, consider the dealer document in Figure 4. A DTD corresponding to it might consist of the rules:

root: *dealer*
dealer $\to$ *UsedCars NewCars*
UsedCars $\to$ *ad**
NewCars $\to$ *ad**
ad $\to$ *model year* | *model*

However, 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, extensions of DTDs provide mechanisms to decouple element names from their types and thus allow context-dependent definitions of their structure. Interestingly, this also leads to closure of the definable sets of trees under Boolean operations. We show one way to formalize the decoupling of names from types, using the notion of *specialized DTD* (studied in [65] and similar to formalisms proposed in [14, 27] and adopted in restricted form in XML-Schema and XQuery). The idea is to use whenever necessary "specializations" of element names with their own type definition. More precisely, a *specialized DTD* for alphabet Σ is a 4-tuple $\langle \Sigma, \Sigma', d, \mu \rangle$ where:
(1) Σ, Σ' are finite alphabets;

(2) d is an DTD over Σ'; and,
(3) μ is a mapping from Σ' to Σ.

Intuitively, Σ' provides for some $a \in \Sigma$, a set of specializations of a, namely those $a' \in \Sigma'$ for which $\mu(a') = a$. Note that μ induces a homomorphism on words over Σ', and also on trees over Σ' (yielding trees over Σ). We also denote by μ the induced homomorphisms. Let us denote specialized DTDs by bold letters **d,e,f**, etc.

Let $\mathbf{d} = \langle \Sigma, \Sigma', d, \mu \rangle$ be a specialized DTD. A tree t over Σ satisfies $\mathbf{d}$ if $t \in \mu(sat(d))$. Thus, t is a homomorphic image under μ of a derivation tree in d. Equivalently, a labeled tree over Σ is valid if it can be specialized to a tree which is valid with respect to the DTD over the specialized alphabet.

For example, we can now write a specialized DTD distinguishing used car ads from new car ads in the dealer example as follows. $\Sigma = \{dealer,\ UsedCars,\ NewCars,\ ad,\ model,\ year\}$, $\Sigma' = \Sigma \cup \{ad^{used}, ad^{new}\}$, μ is the identity on Σ and $\mu(ad^{used}) = \mu(ad^{new}) = ad$ and the DTD over Σ' (with ϵ-rules omitted) is:

root: *dealer*
$dealer \rightarrow UsedCars\ NewCars$
$UsedCars \rightarrow (ad^{used})^*$
$NewCars \rightarrow (ad^{new})^*$
$ad^{used} \rightarrow model\ year$
$ad^{new} \rightarrow model$

Many interesting basic questions arise in connection to XML schemas. 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? We next discuss a powerful, effective tool for dealing with such questions: a remarkable connection between schemas and tree automata.

XML schemas, tree automata, and logic. 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 $(a, q) \rightarrow (q', q'')$, 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 $(a, q) \rightarrow q_f$ 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 nondeterministically 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. Deterministic and non-deterministic bottom-up automata can also be defined, and they are both equivalent to the non-deterministic top-down automata.

There is a strong connection between regular tree languages and logic, similar to the string case. Regular tree languages are precisely those definable by Monadic Second-Order logic (MSO) on structures representing binary trees. Regular languages of finite binary trees are surveyed in [35].

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 [16] (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* [16, 65].

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. Efficient incremental validation of XML documents also makes use of tree automata [66]. Finally, as we will see shortly, tree automata are crucial tools in the static analysis of XML queries.

2.4 Constraints

Constraints are essential ingredients to classical databases. While their primary role is as a filter of invalid data, they are also useful in query optimization, schema design, and choice of efficient storage and access methods. The most common database constraints are *functional dependencies* (fds) and *inclusion dependencies* (incds). Not surprisingly, these continue to be important in semi-structured data and XML. However, the difference in frameworks leads to significant differences in how constraints are specified and their properties.

Constraints in semi-structured data. The constraints that have emerged for semi-structured data are mostly variants of inclusion dependencies. These are expressed using *path constraints*. There can be viewed as logical statements whose atoms are expressions of the form $r(x, y)$ where r is a regular expression over the set Σ of labels of the data graph. Intuitively, $r(x, y)$ states that y can be reached from x by a path whose labels spell a word in r. For example, consider again the relational database in Figure 2 and its representation as a data graph in Figure 3. Suppose we wish to state the inclusion dependency $R[A] \subseteq Q[C]$. In the database, this is easily done using the schema. In the data graph, referring to A and C is done by specifying how they can be reached from the root. This can be done using a path constraint of the form

$$\forall x[database.R.tup.A(root, x) \rightarrow database.Q.tup.C(root, x)]$$

For simplicity, we abbreviate the statement $\forall x[p(root, x) \rightarrow q(root, x)]$ by $p \subseteq q$ (and $p = q$ stands for $p \subseteq q$ and $q \subseteq p$).

There are many other scenarios in which path constraints arise naturally. They may capture, for instance, structural information about a Web site (or a collection of sites) or cached information. For example, consider the two paths:

p1 = CS-Department DB-group Ullman Classes cs345
p2 = CS-Department Courses cs345

It may be the case that starting from some site `Stanford`, the paths $p1$ and $p2$ lead to the same object. Thus, the path constraint $p1 = p2$ holds at site `Stanford`. Similarly, at the site `CS-Department` one could have the constraint

$$\Sigma^* \texttt{ Stanford-CS-Main} = \epsilon$$

stating that all paths starting at site `CS-Department` whose final label is `Stanford-CS-Main` lead back to that site.

The implication problem for path constraints is a core technical issue. For example, testing if a path query p can be replaced by a "simpler" path query q given structural constraints and caching information captured by a set Σ of path constraints amounts to verifying that $\Sigma \models (p = q)$.

For instance, suppose we know that every path ending by label l returns to the source site, i.e. $\Sigma^* l = \varepsilon$. Suppose query $p = (la + lb)^* d$ must be executed at this site. It can be shown that p is equivalent to $(a + b)d$. This query is likely to be simpler than the original; in particular, it is non-recursive and so is guaranteed to terminate.

It turns out that the general implication problem for regular path queries is decidable in 2-EXPSPACE [6]. This is shown by placing a bound of the minimum size of data graphs providing a counter-example to the implication. A more tractable case is that of *word constraints* of the form $u \subseteq v$ where u, v are single words. The implication problem for word constraints is in PTIME, and implication of path constraints by word constraints is in PSPACE. Interestingly, the implication of word constraints can be reduced to testing satisfiability of an FO^2 sentence[1], which is known to be decidable in NEXPTIME [39]. The improved PTIME bound is obtained in [6] by showing that the language $\{v \mid \Sigma \models u = v\}$ is regular and an automaton accepting it can be constructed in PTIME from Σ and u.

It turns out that more complex path constraints are needed in many situations. For example, the paths considered above always start at the root of the data graph. It is useful to also allow defining a limited scope for the constraints by using as root any internal node reachable from the global root by some specified path. This gives rise to constraints $[p \subseteq q]@r$, meaning that $p \subseteq q$ holds from every node reachable from the root by a path in r. Surprisingly, this seemingly benign extension has dramatic impact on the earlier decidability results: the implication problem becomes undecidable even when p, q are words and r is a single letter! (The proof, presented in [22], is by reduction of the word problem for finite monoids.) Such constraints, as well as extensions allowing to express *inverse relationships* (e.g. the *takes* relationship from students to courses is the inverse of the *taken-by* relationship from courses to students) are studied in [21, 22]. The interaction of schemas and constraints is also studied there, and it is shown that schemas have significant impact on the constraint implication problem: some instances of the problem that are decidable in the schema-less case become undecidable when schemas are present, and conversely.

Constraints in XML. Just as in semi-structured data, there is a natural need to express inclusion dependencies in XML documents. In addition, key constraints are part of various schema proposals, such as XML Schema. Both types of constraints also arise in XML documents that are generated from databases.

In XML, both key constraints and inclusion dependencies involve the *data values* associated to the leaves of XML documents (or to values of attributes viewed as leaf elements), whereas in semi-structured data inclusion dependencies refer to the nodes themselves (data values can be easily modeled as nodes, whereas doing this in XML would destroy the tree structure of documents). Inclusion dependencies in XML can be expressed much like in semi-structured data using path expressions, with extensions for the non-unary case [31]. Key constraints can be formalized as a pair $(q, \{p_1, \ldots, p_n\})$ where q and the p_i's are path expressions. Intuitively, q identifies the elements e to which the key constraint applies and $p_1, \ldots p_n$ the nodes whose data values collectively identify each element e. More precisely, if e, f are nodes reachable from the root by paths in q, the node e_i is reachable from e by a path in p_i, the node f_i is reachable from f by a path in p_i, and the values of e_i and f_i are equal, $1 \leq i \leq n$, then e and f are the same node. Note that this definition uses separate notions of value equality and node equality.

The implication problem for key constraints as above is harder than in the relational case, because it involves reasoning about regular path expressions (and recall that equivalence of regular expressions in isolation is already PSPACE-hard [34]). Restrictions on the path expressions leading to an $O(n^2)$ algorithm for testing implication are shown in [18].

There is an intricate interaction between XML constraints and DTDs. As shown in [30], the satisfiability problem for key and foreign key constraints becomes undecidable in the presence of DTDs (and is NP-complete in the unary case), whereas it is trivial in classical databases. Checking consistency of DTDs together with constraints is further studied in [11], and normal forms for XML documents are discussed in [12]. The impact of DTDs and other schema formalisms on constraints is interesting both theoretically and practically, and remains largely unexplored. A survey of constraints in semi-structured data and XML is presented in [20]. Constraints in semi-structured data are also discussed in [2].

3 Queries on the Web

Much of classical database theory revolves around the theory of query languages. In the relational framework, this is solid, familiar ground. Queries are defined as computable, generic mappings from relational databases to relations. A language is complete if it expresses all queries. There is a well-understood hierarchy of languages, ranging from the conjunctive queries, relational calculus and algebra, and Datalog, all the way to complete languages. Relational calculus is so much a standard that it is used as a yardstick, yielding the notion of "relational completeness". Complexity classes provide a language-independent measure for expressiveness.

In the Web scenario, much of this foundation is shaken. The data to be queried is often a moving target, so queries do not always have a well-defined input. There are no well-accepted yardsticks for expressiveness to replace relational completeness and no nice match to query complexity classes. Query languages mix declarative and navigational features, they usually involve limited recursion, and idiosyncratic forms of negation. The expressiveness of the various languages is hard to characterize, since they are sometimes not even closed under composition. In short, we are in for a

[1] FO^2 denotes the FO sentences using only two variables.

challenging but fascinating ride.

As one illustration of the impact of the Web scenario, let us consider query complexity. How to measure the complexity of a query posed against the Web is a puzzling question. In database theory, characterizing a query in complexity-theore-tic terms provides a first-cut at evaluating its difficulty. The first-order queries have complexity LOGSPACE in the size of the database, and this is often considered reasonable. However, this paradigm is unlikely to transfer to the Web. Indeed, it is hard to imagine that a query that takes LOGSPACE (or any other standard complexity bound) in the size of the Web could be considered reasonable. Moreover, if a query is evaluated against the live Web, the cost of accessing and shipping information across the network is paramount. There have been various attempts to develop cost models that take such factors into account. For example, a cost model distinguishing local and remote links is proposed in [51] in conjunction with the language WebSQL.

A more radical proposal was put forward in [7], where it is suggested that the Web is best modeled by an *infinite* graph (where each node has finite out-degree but possibly infinite in-degree), just like computers with potentially very large but finite memory are best modeled by Turing machines with infinite tapes. In this model, exhaustive exploration of the Web is penalized by a non-terminating computation. This draws a sharp distinction between exhaustive exploration of the Web and more controlled forms of computation. Consider a simple model of queries as mappings from the Web (an infinite rooted graph) returning a subset of its nodes. Queries can then be classified into several categories: (i) *finitely computable* queries are always evaluated in finite time on the infinite Web; (ii) *eventually computable* queries are non-terminating queries with possibly infinite answers, and each node in the answer can be output after finite time with no need to backtrack; and (iii) *non-eventually computable queries* (all others).

For example, the following query is finitely computable: *Find all nodes reachable from the root by a path of length at most 3.* The following queries are eventually computable but not finitely computable: (i) *Find all nodes reachable from the root,* and (ii) *Output the root iff it belongs to some cycle.* Note that the latter query always has a finite answer. Nonetheless it is not finitely computable. The following seemingly innocuous query (which also has a finite answer) is not even eventually computable: *Output the root iff it is not referenced by any other node.* It is not clear whether the above classification has a natural finitary analog.

A similar classification can be applied to standard query languages. Relational calculus can express non-eventually computable queries, but a "positive" fragment can be defined that only expresses eventually computable queries. The Datalog$^\neg$ languages yield some surprises: the standard semantics, stratified and well-founded [36], are ill-suited for expressing eventually computable queries, whereas the inflationary semantics [5, 45] turns out to be naturally suited to express such queries, and thus has an advantage over the first two semantics [7].

3.1 Query Languages

The query languages proposed in the context of the Web vary depending on the target data. Some languages are aimed at querying the Web as a whole, based on the hyperlink structure of Web pages. Such languages include WebSQL [51] and W3QL [46]. Other languages are aimed at semi-structured data, such as Lorel [4] and UnQL [19]. StruQL is part of the Strudel Web site management system, and allows defining linked Web pages as views of semi-structured data inputs [32]. A query language for semi-structured data based on the *ambient calculus* (a modal logic for mobile computation) has recently been proposed [23]. There has been a flurry of proposals for XML query languages, including XML-QL [28], XSLT (W3C Web site), XMAS [13], XQL [68], XDuce [41, 42], and Quilt [24]. Recently, the language XQuery has been adopted by the W3C committee as the standard query language for XML. A survey of the query languages for semi-structured data and XML is beyond the scope of this paper (see [1] for a survey on querying semi-structured data).

Several subtleties distinguish XML queries from from semi-structured data. First, it must be ensured that the output is a tree, so care must be taken in how links are specified. Second, the output must be an ordered tree, so mechanisms are needed for specifying the desired order. The order induced on the bindings by the input tree is usually the default. Also, some query languages (e.g. XMAS, YATL) allow querying the order of the input tree, by placing ordering conditions on variables in the pattern bound to sibling nodes, and even using horizontal path expressions among them.

The expressiveness of the query languages for XML and semi-structured data is not easy to characterize. They appear to be a mix of useful but rather ad-hoc constructs, with both declarative and navigational features. They have limited recursion, in the style of Datalog chain queries. The common core is monotonic, but monotonicity is lost under minor variations in the use of regular path expressions. Some variants are not even closed under composition. Thus, the classes of queries expressed by the languages for XML and semi-structured data appear to be rather idiosyncratic and to lack robustness. Nonetheless, we show next that there is a formal framework that convincingly subsumes all of the XML languages: tree transducers.

3.2 XML queries and tree transducers

k-pebble transducers. 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* [52]. This provides a uniform framework for measuring the expressiveness of XML languages, and it is instrumental in developing static analysis techniques. In Section 4 we will see how 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 a fragment of 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. However, k-pebble transducers can be easily extended to handle data values. Details, as well as examples, can be found in [52].

The k-pebble transducers generalize several known formalisms. Aho and Ullman [8] introduce *tree-walking* automata. These devices have a single head which can move up and down the tree, starting from the root. The set of tree languages accepted by a tree-walking automata is included in the set of regular tree languages, but it is a long-standing open problem whether the inclusion is strict [29]. The question whether k-pebble transducers can simulate all bottom-up transducers can be reduced to this open problem (in fact the two problems become equivalent, when $k = 1$). For the case of strings, the analog of tree-walking automata are precisely the two-way automata, which are known to express all regular languages.

String automata with a rather restricted form of k-pebbles are considered by Goberman and Harel [37]. They prove certain lower bounds in the gap of succinctness of the expressibility of such automata. Similarly, it turns out that the emptiness problem for k-pebble automata has a non-elementary lower bound.

Other models. Another transducer model for XML queries, called *query automaton*, is described in [62]. 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 [49], 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 [61].

Another computation model for trees, based on *attribute grammars*, is considered in [59]. 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 [60, 69]. For example, it is shown that this fragment of MSO can express FO extended with regular path expressions. In [15] 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 [16], and has given rise to a fertile line of research into formalisms for unranked trees. This includes extensions of tree transducers [49], push-down tree automata [54], attribute grammars [55], and caterpillar expressions [17]. 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 [44, 63]. Tree-walking transducers accessing data values of XML documents are considered in [58].

XML schema languages contain new constructs allowing to specify flexible order constraints, and in particular to mix ordered and unordered data. XML query languages in turn provide constructs to specify the ordering of nodes in the answer. Neither aspect is captured by traditional tree automata and transducer models.

Other interesting questions involve the processing of XML, including validation with respect to DTDs, and computing queries. Of special interest is the processing of *streaming* XML (e.g., see [43, 48, 70]). Formalizing this would require automata and transducer models that perform a single traversal of the input tree in depth-first, left-to-right order.

XML is already stimulating new research directions in language theory, and this trend is likely to amplify. A successful relationship will be a symbiotic one, in the mold of relational database theory and finite-model theory. Informative surveys on logic and automata-theoretic approaches to XML are provided in [57, 56].

4 Typechecking XML Queries

In relational databases, typechecking is a non-issue: in the standard relational query languages, the schema of the result is apparent from the syntax of the query. The situation is very different for XML. Whether the result of an XML query (or transformation) always satisfies a target DTD is far from obvious. Moreover, this is an important question in many scenarios. A typical one is data integration, where a user community would agree on a common DTD and on producing only XML documents that are valid with respect to the specified DTD.

The *(static) typechecking* problem is the following: given an input XML schema d (e.g., a specialized DTD) a query q, and an output schema d', is it the case that $q(sat(d)) \subseteq sat(d')$?

Related to the typechecking problem is the *type inference* problem[2]: given an input schema d and a query q, compute an output schema $\bar{q}(d)$ for $q(sat(d))$. This can mean several things. If $q(sat(d)) \subseteq sat(\bar{q}(d))$ then the inference algorithm computing $\bar{q}(d)$ is *sound*, and this is clearly a minimum requirement. Ideally, it would also be the case that $q(sat(d)) = sat(\bar{q}(d))$; then the inference algorithm is said to be sound and complete. Note that, in particular, a sound and complete inference algorithm would also solve the typechecking problem. Indeed, to verify that $q(sat(d)) \subseteq sat(d')$ it would be sufficient to check that $sat(\bar{q}(d)) \subseteq sat(d')$, which is decidable.

Unfortunately, sound and complete type inference is not possible for standard XML queries. An approach to incomplete type inference is taken by XDuce [41, 42]. In XDuce, types are essentially specialized DTDs. Recursive functions can be defined over XML data by pattern matching against regular expressions. XDuce performs static typechecking for these functions, verifying that the output of a function will always be of the claimed output type. However, the typechecking algorithm is only sound, not complete: one can write in XDuce a function that always returns results of the required output type, but that the typechecker rejects. This is expected in a general-purpose language that can express non-terminating functions. XDuce focuses on making the typechecker practical, both for the application writer and for the language implementer. A similar approach is taken by YATL [27, 26]. This language for semistructured data has an original type system, based on unordered types. Like XDuce, YATL admits incomplete type inference.

It turns out that sound and complete typechecking can be performed for a wide variety of XML languages so long as they query the tree structure of the input but not its data values. This is explored in [52] using the k-pebble transducer. As discussed earlier, this subsumes the tree manipulation core of most XML languages. Typechekcing 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 [10]. An overview of typechecking for XML is provided in [71].

Another twist in the typechekcing problem arises in the increasingly common scenario of relational databases exporting XML views of the data. Queries are then mappings from relations to trees. For example, SilkRoute is a research prototype enabling the definition of XML views from a relational database [33]. The typechecking problem now asks whether all views generated from the database satisfy a target DTD, possibly specialized. The database itself may satisfy given integrity constraints. This problem is investigated in [9], using an abstraction of the query language of SilkRoute. Once again, the general problem is undecidable, and the limits of decidability are established.

[2] The variant we state differs from that used in programming languages by assuming the input type is given.

5 Conclusion

In order to meaningfully contribute to the formal foundations of the Web, database theory has embarked upon a fascinating journey of rediscovery. In the process, some of the basic assumptions of the classical theory had to be revisited, while others were convincingly reaffirmed. There are several recurring technical themes. They include extended conjunctive queries, limited recursion in the form of path expressions, ordered data, views, incomplete information, active features. Automata theory has emerged as a powerful tool for understanding XML schema and query languages. The specific needs of the XML scenario have in turn provided feedback into automata theory, generating new lines of research.

Many other important areas of research were left out of this incomplete account. Some of these, addressed at length in other surveys, include: data integration, hidden data privacy, protection, cryptography, workflows for interactive web sites, data and schema mining, information retrieval and meta-data.

The Web scenario is raising an unprecedented wealth of challenging problems for database theory – a new frontier to be explored.

6 Acknowledgments

The author is grateful to Serge Abiteboul, Peter Buneman, Frank Neven, Luc Segoufin, Dan Suciu, and Moshe Vardi for useful comments and suggestions on the full version of this paper [73].

References

[1] S. Abiteboul. Querying semi-structured data. In *Proc. ICDT*, pages 1–18, 1997.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan Kauffman, 1999.

[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] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The LOREL query language for semistructured data. *Journal of Digital Libraries*, 1(1), 1997.

[5] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proc. ACM PODS*, pages 240–250, 1988.

[6] S. Abiteboul and V. Vianu. Regular path queries with constraints. *JCSS*, 58(3):428–452, 1999.

[7] S. Abiteboul and V. Vianu. Queries and computation on the Web. *Theoretical Computer Science*, 239(2):231–255, 2000. Extended abstract in ICDT 97.

[8] A. Aho and J. Ullman. Translations on a context free grammar. *Information and Control*, 19(19):439–475, 1971.

[9] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. In *Proc. IEEE LICS*, pages 421–430, 2001.

[10] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: typechecking revisited. In *Proc. ACM PODS*, 2001.

[11] M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *Proc. ACM PODS*, pages 259–270, 2002.

[12] M. Arenas and L. Libkin. A normal form for XML documents. In *Proc. ACM PODS*, pages 85–96, 2002.

[13] C. Baru et al. XML-based information mediation with MIX. In *ACM SIGMOD Conf. Demo.*, pages 597–599, 1999.

[14] 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.

[15] G. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. In *Proc. DOOD*, pages 1137–1151, 2000.

[16] 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.

[17] A. Brüggemann-Klein and D. Wood. Caterpillars: a context specification technique. *Markup Languages*, 2(1):81–106, 2000.

[18] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *Proc. WWW-10*, 2001.

[19] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. ACM SIGMOD*, pages 505–516, 1996.

[20] P. Buneman, W. Fan, J. Simeon, and S. Weinstein. Constraints for semi-structured data and XML. *SIGMOD Record*, 30(1), 2001.

[21] P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured and structured databases. In *Proc. ACM PODS*, pages 129–138, 1998.

[22] P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. In *Proc. ACM PODS*, pages 56–67, 1999.

[23] L. Cardelli and G. Ghelli. A query language based on the ambient logic. In *Proc. European Symp. on Programming*, 2001. Invited paper.

[24] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, pages 53–62, 2000.

[25] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *IPJS*, pages 7–18, 1994.

[26] V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. In *Proc. ACM SIGMOD*, pages 141–152, 2000.

[27] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. ACM SIGMOD Conf.*, pages 177–188, 1998.

[28] A. Deutsch, M. Fernandez, D.Florescu, A. Levy, and D. Suciu. A query language for XML. In *WWW8*, pages 11–16, 1999.

[29] J. Engelfriet, H. Hoogenboom, and J. Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.

[30] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *JACM*, 49(3):368–406, 2002.

[31] W. Fan and J. Siméon. Integrity constraints for XML. In *Proc. ACM PODS*, pages 23–34, 2000.

[32] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Catching the boat with Strudel: experience with a web-site management system. In *Proc. ACM SIGMOD Conf.*, 1998.

[33] M. Fernandez, W. Tan, and D. Suciu. Silkroute: trading between relations and XML. *Computer Networks*, (33):723–745, 2000.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman, 1979.

[35] 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.

[36] A. V. Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:620–650, 1991.

[37] N. Globerman and D. Harel. Complexity results for two-way and multi-pebble automata and their logics. *TCS*, 169(2):161–184, 1996.

[38] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB*, pages 436–445, 1997.

[39] E. Grädel, P. Kolaitis, and M. Vardi. On the complexity of the decision problem for two-variable first-order logic. *Bulletin of Symbolic Logic*, 3(1):53–69, 1997.

[40] J. Hammer et al. Information translation, mediation, and mosaic-based browsing in the TSIMMIS system. In *Proc. ACM SIGMOD Conf.*, page 483, May 1995.

[41] H. Hosoya and B. Pierce. XDuce: A typed XML processing language (Preliminary Report). In *WedDB (Informal Proceedings)*, pages 111–116, 2000.

[42] H. Hosoya, J. Vouillon, and B. Pierce. Regular expression types for XML. In *Int. Conf. on Functional Programming*, pages 11–22, 2000.

[43] Z. Ives, A. Levy, and D. Weld. Efficient evaluation of regular path expressions on streaming XML data. Univ. of Washington Tech. Rep. CSE000502.

[44] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[45] P. G. Kolaitis and C. Papadimitriou. Why not negation by fixpoint? In *Proc. ACM PODS*, pages 231–239, 1988.

[46] D. Konopnicki and O. Shmueli. W3QS: A query system for the World Wide Web. In *Proc. VLDB Conf.*, pages 54–65, Zürich, Switzerland, Sept. 1995.

[47] D. Lee and W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87, 2000.

[48] B. Ludaescher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *Proc. VLDB Conf.*, 2002.

[49] S. Maneth and F. Neven. Structured document transformations based on XSL. In *Proc. DBPL*, pages 79–96. LNCS, Springer, 1999.

[50] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[51] A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. In *Proc. PDIS Conf.*, 1996.

[52] 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.

[53] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *Proc. ICDE Conf.*, 1997.

[54] 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.

[55] F. Neven. Extensions of attribute grammars for structured document queries. In *Proc. DBPL*, pages 97–114. LNCS, Springer, 2000.

[56] F. Neven. Automata, logic, and XML. In *Proc. Computer Science Logic*, pages 2–26. Springer LNCS, 2002.

[57] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[58] F. Neven. On the power of walking for querying tree-structured data. In *Proc. ACM PODS*, pages 77–84, 2002.

[59] 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.

[60] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. ACM PODS*, pages 145–156, 2000.

[61] F. Neven and T. Schwentick. On the power of tree-walking automata. In *Proc. ICALP*, pages 547–560, 2000.

[62] F. Neven and T. Schwentick. Query automata on finite trees. *Theoretical Computer Science*, 275(1-2):633–674, 2002.

[63] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. In *Proc. MFCS*, pages 560–572, 2001.

[64] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. ICDE Conf.*, pages 251–60, 1995.

[65] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. ACM PODS*, pages 35–46, 2000.

[66] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proc. ICDT Conf.*, pages 47–63, 2003.

[67] P.Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proc. DBPL*, 1995.

[68] J. Robbie, J. Lapp, and D. Schach. XML query language (XQL). In *The Query Languages Workshop (QL'98)*, 1998.

[69] T. Schwentick. On diving in trees. In *Proc. MFCS*, pages 660–669, 2000.

[70] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proc. ACM PODS*, pages 53–64, 2002.

[71] D. Suciu. The XML typechecking problem. *SIGMOD Record*, 31(1):89–96, 2002.

[72] J. Thierry-Mieg and R. Durbin. Syntactic definitions for the ACeDB data base manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge, UK, 1992.

[73] V. Vianu. A Web odyssey: From Codd to XML. In *Proc. ACM PODS*, pages 1–15, 2001.