A Web Odyssey: from Codd to XML

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

What does the age of the Web mean for database theory? It is a challenge and an opportunity, an exciting journey of rediscovery. These are some notes from the road.

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 will discuss 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

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

PODS '01 Santa Barbara, California USA
Copyright 2001 ACM 0-89791-88-6/97/05 ...$5.00.

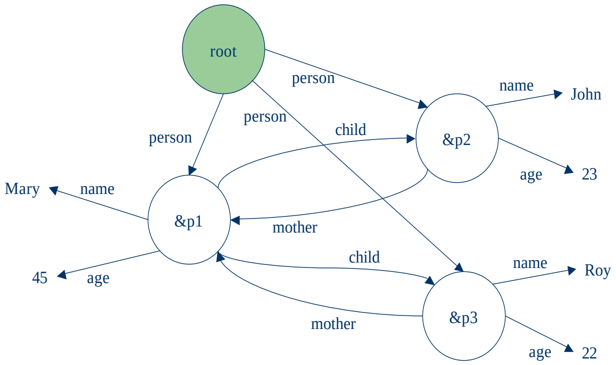


Figure 1: A semi-structured data graph

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 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.

Database theory has started to tackle the Web, but this is only the beginning. There is a wide range of Web applications that provide a new frontier for database theory. Here are some examples. Managing integrated views of multiple data sources has raised technical issues including *view-based query answering*, also related to query optimization and semantic caching. The emergence of Web communities has raised the need to establish and use ontologies and meta-data. Related questions are raised by the need to deal with non-traditional sources that can be only be accessed by *limited patterns*. A special challenge is raised by *interactive* Web sites, such as those commonly arising in e-commerce applications. These can be viewed as sources of information, but also as special forms of active databases implementing a *workflow* modeling the interaction with customers or other sites. Repositories of Web data must cope with the highly dynamic nature of the Web, and must deal with temporal aspects related to data freshness, consistency, incomplete information, and push and pull technology. Search engines use a mix of database and information retrieval techniques. The interplay of the two is one of the most challenging issues yet to be addressed.

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. We revisit the classical issues of genericity, order, and query complexity. Then we present the main approaches to querying the Web, semi-structured data, and XML. Section 4 discusses query typechecking. New frontiers are discussed in Section 5, and the last section provides some conclusions.

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.

Yet there is another side to the story. The Web is a moving target, but it is not arbitrary. As described by Raghavan [71], the graph structure of Web pages and their hyperlinks exhibits interesting and fairly stable structural properties that could be exploited. Search engines use structural properties to find authoritative Web sites. Identifying and taking advantage of the stable patterns in the changing Web is a novel and exciting prospect from a database perspective.

There are more tractable alternatives to viewing the entire Web as a single database. Much of the data exchange on the Web takes place in more controlled environments. At one end of the spectrum, many data-intensive Web sites are really classical databases with XML wrappers. Next, there are Web sites that provide *integrated views* of a collection of distributed data sources. The collection could be statically defined or dynamic, but often limited to some community of users explicitly or implicitly sharing an ontology and the metadata for using it. At the other end of the spectrum are servers that aim to extend search engines by answering queries as best possible on the entire Web, or perhaps on the XML-ized portion of the Web, by a centralized or decentralized approach.

R	A	B	C
	1	1	2
	2	1	3

Q	C	D
	2	1
	1	0

Figure 2: A relational database

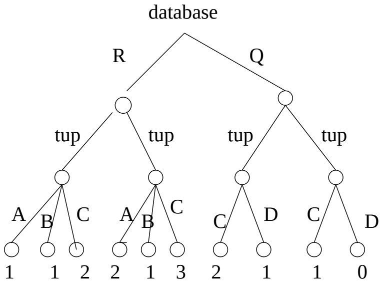


Figure 3: Data graph for R and Q

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 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 first semi-structured data model was the Object Exchange Model (OEM), introduced within the Tsimmis project as a vehicle for integrating heterogeneous sources [58, 34, 91]. This was soon followed by Lore [6, 75]. Another model, UnQL, was developed at the University of Pennsylvania [93, 25], motivated by the OEM model and by the ACeDB graph model used in biological databases [98].

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).

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

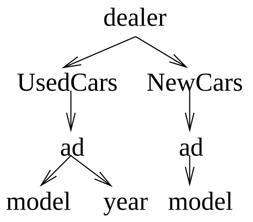


Figure 4: Dealer XML document

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.

A schema for semi-structured data constrains the *paths* (more precisely the sequences of labels along paths) that can be found in the data graph. This is the natural extension to semi-structured data of relational or object-oriented schemas. For example, this allows specifying that the paths found in data graphs representing relations R, Q (Figure 3) are precisely $\{\mathit{database.R.tup.A}$, $\mathit{database.R.tup.B}$, $\mathit{database.R.tup.C}$, $\mathit{database.Q.tup.C}$, $\mathit{database.Q.tup.D}\}$ (note that this is not quite equivalent to the relational schema since it does not ensure uniqueness of the attributes for each tuple). More generally, it is useful to be able to specify both paths that *must* be found in the data and paths that *may* be present but are not required. This leads to two kinds of schemas: *lower-bound* and *upper-bound* schemas. Both can be specified by graphs summarizing the path information, with semantics based on the notion of *simulation.* Consider a data graph D and another edge-labeled graph G that is used as a schema. A simulation from G to D is a relation R from the vertices of G to those of D such that whenever xRu and there is an edge labeled a from x to y in G, there exists v in D so that $y\ R\ v$ and there is an edge labeled a from u to v in D. It turns out that simulations can be computed very efficiently [59].

A data graph D satisfies the schema graph G as a lower bound if there exists a simulation from G to D. In particular, this ensures that every sequence of labels found on a path in G is also found in D. Similarly, D satisfies G as an upper bound if there exists a simulation from D to G. This in turn ensures that every path of D occurs in G. To allow more flexibility, upper bound schemas have been enriched with disjunctions of labels and wildcards (allowing any label).

In the typical scenario, schemas for semi-structured data are not provided *a priori* but are instead *extracted* from the data. Both maximal lower-bound and minimal upper-bound schemas can be extracted from data graphs, as well as schemas summarizing *precisely* the paths in the data (e.g., the *data guides* of [53]). Clearly, there is a trade-off between accuracy and conciseness of extracted schemas. The data graph itself can serve as both lower and upper bound schemas, but this is not satisfactory. Coming up with the right middle ground is an interesting and largely open problem.

An interesting problem related to schema extraction is *classification* of the nodes in a data graph. Suppose we have a lower-bound schema G for a data graph D. It is natural to define the classes as the nodes of G and to place in a class c all nodes o of D for which cRo holds in the maximum simulation R from G to D (which always exists). Interestingly, equivalent classifications can be defined using Datalog programs with greatest-fixpoint semantics.

Upper bound schemas were first defined in [24] as *graph schemas.* The data guides of [53] were anticipated by the *representative objects* of [81]. The Datalog approach to typing and classification is explored in [80]. The connection between Datalog and simulation is a special case of a more general connection between Datalog and asymmetric pebble games [68].

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 [72].

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.

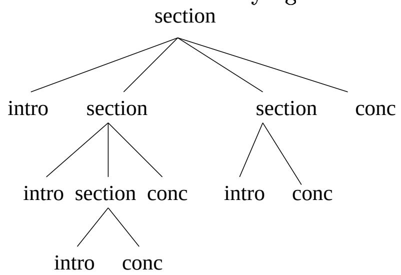
Essentially, a DTD is an extended context-free grammar. The non-terminals of the grammar are the labels (tags) of elements in the labeled tree corresponding to the XML document. There are no terminal symbols.

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: $dealer$

$dealer \to UsedCars\ NewCars$

$UsedCars \to ad^*$

$NewCars \to ad^*$

$ad \to model\ year \mid model$

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 [92] and equivalent to formalisms proposed in [17, 37] and in XML-Schema). 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*
$dealer \rightarrow UsedCars\ NewCars$
$UsedCars \rightarrow (ad^{used})^*$
$NewCars \rightarrow (ad^{new})^*$
$ad^{used} \rightarrow model\ year$
$ad^{new} \rightarrow model$

A less powerful alternative to specialization is to make type specification dependent on a context specified by a path expression. This is done, for example, in XDuce [60, 61]. Such extensions alleviate some limitations of DTDs, but there are many others. Conspicuously missing is a *subtyping mechanism*, a drawback partially remedied in later proposals such as XML-Schema (as well as XDuce). Another interesting aspect is the specification of *ordering constrains* among children of a given element. DTDs use regular expressions for this purpose, and this has several drawbacks, of which we discuss two: excessive power and limited flexibility. On one hand, regular expressions may be too powerful for most practical situations. For example, they allow stating properties unlikely to be useful, such as *"the number of UsedCar ads must be even"*. More restricted formalisms, such as the *star-free* regular languages, are often sufficient. To understand the significance of such restrictions, it is useful to consider a logic-based point of view. First, note that strings over alphabet Σ can be viewed as logical structures over the vocabulary $\{<, (O_\sigma)_{\sigma \in \Sigma}\}$ where $<$ is a binary relation and every O_σ is a unary relation. A string $w = a_1 \ldots a_n$ is represented by the logical structure $(\{1, \ldots, n\}; <, (O_\sigma)_{\sigma \in \Sigma})$ where $<$ is the natural order on $\{1, \ldots, n\}$, and for each i, $i \in O_\sigma$ iff $a_i = \sigma$. It is well-known that regular languages are exactly those definable by Monadic Second-Order (MSO) logic[1] on the logical vocabulary of strings [22, 41]. However, this is much more powerful than needed by most DTDs. In many cases, the required properties of valid strings can be expressed simply in First-Order logic (FO). This corresponds to a well-known subset of the regular languages, called *star-free* [99]. There is a language-theoretic characterization of star-free languages: they are precisely described by the *star-free regular expressions*, which are build from single symbols and ϵ using concatenation, union, and complement. Another nice formalism equivalent to FO on strings is *propositional temporal logic over finite words* [65]. In temporal logic, one can make statements such as "each *name* is immediately followed by the *ssn*, eventually followed by *city*, *street*, and *zip* in some order; a *phone* and *email* are optional". These are very intuitive, commonly arising statements and are often sufficient. Without going into details, it turns out that replacing regular languages by star-free languages decreases the complexity of various problems related to typing.

Another limitation of DTDs arising from the use of regular expressions is the lack of flexibility in specifying ordering constraints. For example, to state that $a_1, \ldots, a_n$ occur in any order, one has to write a disjunction of $n!$ expressions $a_{\sigma(1)} \ldots a_{\sigma(n)}$ where σ is a permutation of $\{1, \ldots, n\}$. Some extensions of DTDs, such as XML Schema, allow direct specification of such constraints. However, the impact of the enriched syntax on the complexity of manipulations is yet unclear (see [66] for results on the succinctness of unordered concatenation such as above).

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 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. 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. As discussed above, regular languages on strings are precisely those definable by Monadic Second-Order logic (MSO) on the structures representing strings in a standard way. This can be extended to trees: regular tree languages are precisely those definable by MSO on structures representing binary trees. Regular languages of finite binary trees are surveyed in [50].

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 [20] (an extension for unranked infinite trees is described in [5]). Either way, one can

[1] MSO is first-order logic augmented with quantification over sets.

prove a surprising and satisfying connection between specialized DTDs and tree automata: *they are precisely equivalent* [20, 92].

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.

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 [9]. 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[2], which is known to be decidable in NEXPTIME [54]. The improved PTIME bound is obtained in [9] 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 [28], 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 [27, 28]. 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 dependen-

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

cies in XML can be expressed much like in semi-structured data using path expressions, with extensions for the non-unary case [44]. 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 [47]). Restrictions on the path expressions leading to an $O(n^2)$ algorithm for testing implication are shown in [23].

There is an intricate interaction between XML constraints and DTDs. As shown in [43], 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. 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 [26]. 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 challenging but fascinating ride.

3.1 Back to Basics

Let us first revisit some of the basic elements in the classical theory of query languages.

Data independence and genericity. Perhaps the single most significant distinguishing characteristic of a database is *data independence*, the separation of the logical and physical levels of data. A direct consequence is that queries are *generic*: their answers depend only on the logical level of data. More formally, queries commute with isomorphisms of the input.

On the Web, the distinction between logical and physical levels is much less clear. Are URLs logical, or physical? Queries certainly do not treat them equally, and we would not want them to. Does the display of information in a Web page convey logical information, or is it purely physical? How about the geographical location of the Web site? Unfortunately there is no single, clear-cut answer. As a corollary, query genericity is also a much less robust and useful notion.

Order. The issue of order plays a central role in the theory of query languages. Databases are typically unordered, and the lack of order has dramatic impact on query language expressiveness. As one example, on ordered databases the *fixpoint* queries express exactly PTIME ([100, 62], see also [4]). In the absence of order, *fixpoint* cannot express even simple queries like the parity of a set. In fact, it is conjectured that there is *no* language expressing PTIME [56].

Data on the Web, as viewed by queries, seems to be in many cases naturally ordered. Queries against the Web as a whole tend to be navigational. They explore the Web following hyperlinks from significant entry points (which might be fixed or obtained by search engines). This induces an ordering on the data. In more limited scenarios, the target of the query may be an XML document, which is again ordered. The presence of order is mixed news. In principle, it should be good news for query language expressiveness, although this remains to be demonstrated. But unordered data has some advantages that may be lost, including better potential for query optimization and parallel evaluation.

Since databases are unordered and XML is ordered, this leads to a replay of the infamous *impedance mismatch* between database query languages and general programming languages [15]. When databases are exported as XML views, the order in the XML documents must be made up in some arbitrary fashion, not determined by the database. This may lead to inconsistencies when the XML view of the database is consumed by XML queries, which generally assume the order has semantic significance. Eliminating this mismatch would require enriching the XML model to accommodate a mix of ordered and unordered data.

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-theoretic 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 [76] in conjunction with the language WebSQL.

A more radical proposal was put forward in [10], 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.

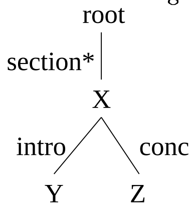
A similar classification can be applied to standard query languages. Relational calculus can express non-eventually computable queries, but a "positive" fragment can be defined that only expresses eventually computable queries. The Datalog$^{\neg}$ languages yield some surprises: the standard semantics, stratified and well-founded [51], are ill-suited for expressing eventually computable queries, whereas the inflationary semantics [8, 69] turns out to be naturally suited to express such queries, and thus has an advantage over the first two semantics [10].

3.2 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 [76] and W3QL [70]. Other languages are aimed at semi-structured data, such as Lorel [6] and UnQL [25]. StruQL is part of the Strudel Web site management system, and allows defining linked Web pages as views of semi-structured data inputs [45]. A query language for semi-structured data based on the *ambient calculus* (a modal logic for mobile computation) has recently been proposed [32]. There has been a flurry of proposals for XML query languages, including XML-QL [40], XSLT (W3C Web site), XMAS [16], XQL [95], XDuce [60, 61], and Quilt [33].

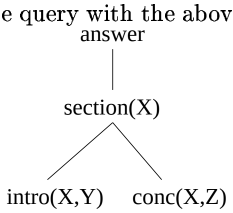
A survey of the query languages for semi-structured data and XML is beyond the scope of this paper (see [1] for a survey on querying semi-structured data). Query languages for XML are in a state of flux, and there is no definitive winner so far[3]. However, two fairly stable paradigms seem to emerge across various languages, well illustrated by XML-QL (or Lorel) and XSLT.

In the first approach, a query consists of two parts: (i) a *pattern* used to extract bindings for a set of variables, and (ii) a *construct* clause indicating how to build the answer from the set of bindings found in (i). The pattern in (i) is in the spirit of conjunctive queries, except more navigational in flavor and extended with limited recursion. The pattern can be viewed as a "map" indicating how to reach the variables from the root or from each other by regular path expressions. Thus, the pattern is a graph of variables strongly connected to the root and labeled by regular path expressions. In the case of XML, the graph is a tree and the data values associated to nodes can be explicitly compared, allowing to perform data joins (recall that in the semi-structured model data values are also nodes, so the distinction is not needed). Variables in the pattern bind to nodes in the input that can reach each other by paths matching the regular expressions in the pattern. For example, consider again XML documents described by the DTD $\begin{array}{l} \mathit{root} : \mathit{section}; \\ \mathit{section} \rightarrow \mathit{intro}, \mathit{section}^*, \mathit{conc} \end{array}$
A query against such documents might use the pattern



The variable X binds to sections, Y binds to X's introduction, and Z to X's conclusion.

The *construct* clause specifies how to build the answer from the set of bindings. In languages for semi-structured data, the output is a labeled graph; in languages for XML, it is a labeled tree. Thus, the *construct* clause provides, in languages for XML or semi-structured data, a way to specify the nodes in the answer, as well as the links between the nodes and the (node or edge) labels. For example, a construct clause for the query with the above pattern might be:



This specifies an answer as follows. One node $section(X)$ is created for each binding of X. For each binding of X, Y, a node labeled $intro(X, Y)$ is created, and for each binding of X, Z a node $conc(X, Z)$ is created. (The expressions *section*(X), *intro*(X,Y), *conc*(X,Z) are referred to as *Skolem functions*, see [2].) For each binding of X, Y, Z, the nodes are linked as specified. So the answer consists of the unnested sequence of sections, and for each section its introduction and conclusion. Some languages, such as XML-QL, allow nesting of queries within the *construct* clause.

Several subtleties distinguish XML 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.

[3] The most recent XML query language proposal from W3C is X-Query (see the W3C Web site).

In the languages described above, the depth of nesting of the *construct* clause is statically determined by the syntax of the query. This is a serious limitation. For example, one cannot make a simple change to an XML document, such as replacing all *name* labels occurring under *person* by *pname*, without knowing its structure. Intuitively, such a query has the flavor of a tree *transformation*, where the output is obtained by modifying recursively the input tree. The language XSLT allows defining such transformations (and much more complex ones!) using structural recursion on trees. For example, (using the notation $a(t_1, t_2)$ to denote a tree with root a and subtrees t_1, t_2), the above transformation can be expressed on binary trees as the function f defined by structural recursion as follows:

$$\begin{array}{rcl} f(x(t_1,t_2)) & = & \text{if } x \neq \mathit{person} \text{ then } x(f(t_1), f(t_2)) \\ & & \text{else } x(g(t_1), g(t_2)) \\ g(x(t_1,t_2)) & = & \text{if } x \neq \mathit{name} \text{ then } x(g(t_1), g(t_2)) \\ & & \text{else } \mathit{pname}(g(t_1), g(t_2)) \end{array}$$

The expressiveness of the query languages for XML and semi-structured data is not easy to characterize. Languages in the style of XML-QL appear to be a mix of useful but rather ad-hoc features. They have limited recursion, in the style of Datalog chain queries. They mix declarative and navigational features. The common core is monotonic, but monotonicity is lost under minor variations in the use of regular path expressions. Their data complexity is polynomial, but some variants are not even closed under composition. For example, suppose the language has horizontal path expressions. Consider the DTD $root \to a^*$, $a \to b^*$. Consider the query on trees satisfying the DTD, asking whether the total number of b's in the input is even. This is not expressible by a single query in the languages described. However, it is the composition of two queries: the first extracts all b's under a new root. The second checks the parity of the sequence of b's using a horizontal path expression.

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.

3.3 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* [78]. 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 (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 [78].

What is the data complexity of k-pebble transducers? In the case of a deterministic transducer T, there exists a PTIME algorithm that computes a representation of $T(t)$ for an input tree t. It is easy to see that the actual output $T(t)$ can have size exponential in t. Still, the algorithm will produce a polynomial-size encoding of $T(t)$, as a DAG. In the case of non-deterministic k-pebble transducers we need to be more careful what the PTIME data complexity means. In particular T can produce an infinite set of outputs for a given t. It can be shown, however, that for each input tree t, (1) the set $T(t)$ is a regular tree language, and (2) one can construct in PTIME (in the size of t) a tree automaton A_t that accepts the language $T(t)$.

The k-pebble transducers generalize several known formalisms. Aho and Ullman [12] 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 [42]. 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 [52]. 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 [85]. 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 [74], 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 [87].

Another computation model for trees, based on *attribute grammars*, is considered in [84]. 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 [86, 96]. For example, it is shown that this fragment of MSO can express FO extended with regular path expressions. In [18] 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 [20], and has given rise to a fertile line of research into formalisms for unranked trees. This includes extensions of tree transducers [74], push-down tree automata [82], attribute grammars [83], and caterpillar expressions [21]. 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 [64, 88]. The k-pebble transducer can be easily extended with tests on data values, corresponding to the data joins in most XML query languages. 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 [63]). 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.

4. TYPECHECKING XML QUERIES

In relational databases, typechecking is a non-issue[4] : 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 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[5]: 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. For example, consider again the input DTD

$root : section;$
$section \rightarrow intro, section^*, conc$

and the query that collects all the leaves of input documents. The output consists of the strings of well-balanced parenthesis where *intro* is the open and *conc* the closed parenthesis. This is not a regular language, so cannot be specified by a DTD. If DTDs are extended with specialization and the ability to specify the content of elements by context-free

[4] However, typing *polymorphic* relational algebra is far from trivial, see [39].

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

languages, then sound and complete type inference can be achieved in restricted cases. For example, it is shown in [92] that this can be done for XML-QL style queries without data joins, limited to selection of subtrees from the input. This can be extended to a sound but incomplete inference algorithm for queries with more complex constructed answers.

Another approach to incomplete type inference is taken by XDuce [60, 61]. 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 [37, 36]. 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 [78] 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. One such class consists of the queries where all equality tests performed are independent of each other. Consequently, all truth assignments to the equality tests are consistent. As far as typechecking is concerned, the actual equality tests can therefore be replaced by nondeterministic guesses of their truth value, without the risk of inconsistent guesses. A comprehensive study of typechecking in the presence of data values is provided in [14], where a fairly tight boundary of decidability is traced.

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 [46]. 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 [13], using an abstraction of the query language of SilkRoute. Once again, the general problem is undecidable, and the limits of decidability are established.

5. NEW FRONTIERS

As we have seen, database theory has made a good start in providing foundations for semi-structured data, XML, query languages, schemas, constraints, and typechekcing. But this is only the beginning. Many Web applications and scenarios remain to be tackled, and provide a new frontier for database theory. We briefly discuss some of them.

Data integration. Providing integrated access to multiple data sources is a long-standing problem that has again assumed central importance in many Web applications. To begin, a common schema for the integrated data is chosen. Then the connection between the sources and the integrated data must be established. This may be done automatically, using classification techniques such as described in Section 2.3 [80]. Alternatively, the connection may be explicitly defined. There are two main ways to do this. First, the integrated data can be defined as a view of the sources. Queries posed against the integrated view must then be translated into queries against the sources. However, this is not always easy to do efficiently if the view is not materialized. Instead, the relationship between sources and integrated data can be turned on its head: each source can be defined as a view of the (virtual) integrated data. The problem of answering a query can then be elegantly modeled as one of *view-based query answering.* Answering a query amounts to rewriting it using the views. This is a well-studied problem, of interest in many applications ranging from query optimization to caching. It has been mostly investigated and largely solved for relational databases, and views and queries defined by conjunctive queries (see the survey [57]). In the framework of data integration, there are specific difficulties. First, the sources may have limited capabilities, including restricted access patterns requiring that some attributes be provided before others can be accessed [94]. Second, the requirement that the rewritten query be equivalent to the original can often be relaxed: it is sufficient if the rewritten query provides a subset of the real answer, preferably maximal. Finally, query rewriting for semi-structured data and XML is largely unexplored. Complexity results on query rewriting for queries and views defined by regular path expressions are provided in [29, 31, 30].

An interesting approach to the problem of view-based query answering is in terms of *incomplete information* [3, 55]. The views provided by the sources form a representation system for incomplete information: a set of views represents the databases in their pre-image. It is then possible to define the *certain* answers to a query q. A tuple t is in the certain answer to q given a set V of views, if $t \in q(D)$ for all databases D in the pre-image of V. Othogonally, one can make a closed-world or an open-world assumption on the sources. Under the open world assumptions, each source contains only a subset of the tuples in the view defining it; under the closed-world assumption, it contains all tuples in the view.

Hidden data. Not all Web sources mean to explicitly export their data. A lot of valuable information is hidden on Web sites behind restricted interfaces. These may consist of forms accurately modeled by limited access patterns (such as those studied in [94, 38]), but may also involve a more complex application-specific protocol. Extracting such data automatically is a difficult task. It requires developing formalisms for specifying the operational aspect of Web applications and using the specifications to generate evaluation plans for queries. Beyond queries, integrated applications, such as comparative shopping, take data integration one step further by bringing a workflow component into the picture. Some recent research has started tackling such problems (e.g., see [77]).

Privacy, protection, cryptography. The converse to accessing hidden data is *protecting* data as it is transferred or processed by third-party Web servers, as well as the *privacy* of users, including their identity and the data they access. This brings to the fore the largely unexplored issue of integrating database technology and cryptography in the context of the Web.

Workflows for interactive web sites. Many interactive, data-intensive applications are governed by intricate workflows that are of interest in their own right. These include e-commerce, digital government, Web-based collaboration, scientific data sources, etc. For example, e-commerce applications use "business models" to specify a protocol of exchanges among partners to a transaction. Typically, this occurs in a data-intensive fashion, with many agents interacting with a Web site simultaneously. It therefore makes sense to approach such applications with a database lens, in order to integrate the data and workflow aspects. In this light, business models are reminiscent of active databases. They can be programmed in a similar manner, and may be amenable to static analysis.

One formal model that captures the interactive Web site scenario is the *relational transducer* [11]. In this model, the state of the application is described by a relational database. The interaction from the outside world is captured by a sequence of input relations. The application responds by a sequence of output relations. Thus, the model can be viewed as a machine that translates an input sequence of relations into an output sequence of relations. For example, consider an e-commerce site where a customer interacts with the site by two input predicates, $order(x, y)$ and $pay(x, y)$. Catalog information about product price is provided by a relation $price(x, y)$. The system responds to inputs with output predicates $sendbill(x, y)$ and $deliver(x)$. In the process it may consult relation *price*, and update the state information. A *run* of a transducer consists of a sequence of inputs and the sequence of outputs generated in response to each of the inputs.

The static analysis of relational transducers is studied in [11, 97]. For instance, *goal reachability* asks if some goal can be achieved by some run of the transducer, possibly with some preconditions. In the example, one might wish to verify that it is possible to achieve the goal $deliver(x)$ as long as $\exists y \; price(x, y)$ holds in the database. In general, however, the problem can be much more complicated. A question of a slightly different flavor is verifying temporal properties satisfied by *all* runs. For instance, the supplier may wish to verify that a product is never delivered before it has been paid. Such questions turn out to be decidable for restricted relational transducers. However, this is just a first step in exploring this multi-faceted topic. One interesting, more complex scenario arises from the interaction of multiple sites, each governed by its own business model.

Data and schema mining. Mining is a useful approach when dealing with large collections of data holding information of potential interest that needs to be discovered. This makes the Web a prime candidate for mining. In the context of the Web, different flavors of mining come up naturally, depending on the focus of the application. *Data mining* may concern the patterns of hyperlinks among Web pages, used to identify authorities, hubs, or Web communities (see the survey [67]). Other variants of data mining may involve identifying sites of interest to a particular topic or application [19]. Data mining on the Web may involve sophisticated algorithms and techniques from information retrieval and machine learning. *XML schema mining* arises as data in various formats is wrapped to produce large collections of XML data. These collections need DTDs, that may be hard to extract manually. An alternative is to mine candidate DTDs from the collection. This raises difficult questions of balancing accuracy and conciseness among the many possible DTDs. As a simple example, suppose the data consists of a finite set of words $\{w_1, \ldots, w_n\}$. There is always an exact description of the set as a regular expression: $w_1 + \ldots + w_n$. However, this is too large to be practical. The description should be relaxed at the cost of allowing some words not in the language. However, the criteria for a good compromise are far from clear. The mining of regular expression patterns is discussed in [49] (see also the survey [48]).

Querying the XML-ized Web. With the emergence of XML as the likely standard for data representation and exchange on the Web, there is increased interest in services aimed specifically at the collection of XML documents on the Web. One approach to querying the XML-ized Web is in a decentralized fashion, exemplified by the Niagara project at the University of Wisconsin [35]. This approach raises very interesting questions related to the distributed, agent-based evaluation of Web queries. An alternative approach is the centralized one, whereby the XML data on the Web is collected and queries in a central repository. This is exemplified by the Xyleme project at INRIA, recently spun off as a start-up (see [89, 90]). The centralized approach raises complex problems related to refresh policies, answering queries in the presence of partially stale data, etc. On the other hand, it also provides opportunities for sophisticated services such as *temporal querying* of XML documents (with versions), querying *changes* to XML documents, subscription services, and making use of push and pull technologies. The problem of detecting and managing change is a difficult one. Besides the algorithmic, indexing, and storage aspects, this is complicated by the lack of uniform semantics attached to the syntax of XML documents. When does a change in the ordering of elements signify a meaningful change in an XML document? Unfortunately, there is no uniform answer. In some cases order is semantically crucial, while in others it is an accident. The latter can arise due to the impedance mismatch between unordered databases and ordered XML, as wrappers generate arbitrary orderings for XML documents representing unordered database.

Since the information in a Web repository is never complete, one interesting problem is the representation and querying of XML documents with *incomplete information*. This is explored in [7]. Incomplete information is also useful in other contexts, such as semantic caching.

Information retrieval and meta-data. The Web is by no means the exclusive domain of databases. To the contrary, other paradigms such as information retrieval (IR) are tough competitors and play a central role in extracting information from the Web. Search engines use a mix of database and IR techniques, but within a very limited framework. The integration of the IR and database paradigms in the context of the Web remains an essential goal.

Meta-data, such as ontologies associated with specific application domains, may provide a welcome bridge between IT and databases. Ontologies provide structure and limits missing from IR at large, but essential to the database paradigm. Technically, they bring to the fore *description logics*, used to effectively specify and reason about classifications and properties of objects. Description logics are fragments of FO, some included in FO^2. It is well-known that FO^2 has many nice properties that the full FO lacks, such as decidability of satisfiability [79]. This explains why reasoning with ontologies can be tractable. A survey of description logics is provided in [73].

6. 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.

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

7. ACKNOWLEDGMENTS

The author is grateful to Serge Abiteboul, Peter Buneman, Frank Neven, Luc Segoufin, Dan Suciu, and Moshe Vardi for useful comments and suggestions.

8. 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 O. Duschka. Complexity of answering queries using materialized views. In *Proc. ACM PODS*, pages 254–263, 1998.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. *JACM*, 45(5):798–842, 1998. Extended abstract in SIGMOD'89.

[6] 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.

[7] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. In *Proc. ACM PODS*, 2001.

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

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

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

[11] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000. Extended abstract in PODS 98.

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

[13] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases, 2001. Manuscript.

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

[15] M. P. Atkinson et al. The object-oriented database system manifesto. In *Proc ACM SIGMOD*, page 395, 1990.

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

[17] 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.

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

[19] S. Brin. Extracting patterns and relations from the world wide web. In *WebDB (Informal Proceedings)*, pages 172–183, 1998.

[20] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree languages over non-ranked alphabets, 1998. Unpublished manuscript.

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

[22] J. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

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

[24] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proc. Int. Conf. on Database Theory*, pages 336–350, 1997.

[25] 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.

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

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

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

[29] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Vardi. Rewriting of regular expressions and regular path queries. In *Proc. ACM PODS*, pages 194–204, 1999.

[30] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Vardi. View-based query processing and constraint satisfaction. In *Proc. IEEE LICS*, pages 361–371, 2000.

[31] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Vardi. View-based query processing for regular path queries with inverse. In *Proc. ACM PODS*, pages 58–66, 2000.

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

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

[34] 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.

[35] J. Chen, D. DeWitt, F. Tian, and Y. Wang. Niagracq: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD*, pages 379–390, 2000.

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

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

[38] H. Davulcu, J. Freire, M. Kifer, and I. Ramakrishnan. A layered architecture for querying dynamic web content. In *Proc. ACM SIGMOD*, pages 491–502, 1999.

[39] J. V. den Bussche and E. Waller. Type inference in the polymorphic relational algebra. In *Proc. ACM PODS*, pages 80–90, 1999.

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

[41] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer Verlag (Second Edition), 1999.

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

[43] W. Fan and L. Libkin. On XML integrity constraints in the presence of dtds. In *Proc. ACM PODS*, 2001.

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

[45] 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.

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

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

[48] M. Garofalakis, R. Rastogi, S. Seshadri, and K. Shim. Data mining and the web: Past, present and future. In *ACM Workshop on Web Information and Data Management (WIDM)*, pages 43–47, 1999.

[49] M. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *Proc. VLDB*, pages 223–234, 1999.

[50] 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.

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

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

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

[54] 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.

[55] G. Grahne and A. Mendelzon. Tableau techniques for querying information sources through global schemas. In *Int'l. Conf. on Database Theory*, pages 332–347, 1999.

[56] Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[57] A. Halevy. Theory of answering queries using views. *SIGMOD Record*, 29(4):40–47, 2000.

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

[59] M. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *Proc. IEEE FOCS*, pages 453–62, 1995.

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

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

[62] N. Immerman. Relational queries computable in polynomial time. *Inf. and Control*, 68:86–104, 1986.

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

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

[65] J. Kamp. Tense logic and the theory of linear order, 1971.

[66] P. Kilpel. SGML and XML content models. *Markup Languages*, 1(2):53–76, 1999.

[67] J. Kleinberg. Hubs, authorities, and communities. *Computing Surveys*, 31(4es), 1999.

[68] P. Kolaitis and M. Vardi. Conjunctive-query containment and constraint satisfaction. *JCSS*, 61(2):302–332, 2000.

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

[70] 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.

[71] S. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. The Web as a graph. In *Proc. ACM PODS*, pages 1–10, 2000.

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

[73] M. Lenzerini. Description logics and their relationships with databases. In *Int'l. Conf. on Database Theory*, pages 32–38, 1999.

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

[75] 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.

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

[77] T. Milo and A. Eyal. Integrating and customizing e-commerce applications. In *VLDB Workshop on Technologies for E-Services, Cairo*, 2000.

[78] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proc. ACM PODS*, pages 11–22, 2000.

[79] M. Mortimer. On languages with two variables. *Zeitschr. f. math. Logik u. Grundlagen d. Math*, 21:135–140, 1975.

[80] S. Nestorov, S. Abiteboul, and R. Motwani. Infering structure in semistructured data. *SIGMOD Record*, 26(4):39–43, 1997.

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

[82] 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.

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

[84] F. Neven and J. V. den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *Proc ACM PODS*, pages 11–17, 1998.

[85] F. Neven and T. Schwentick. Query automata. In *Proc. ACM PODS*, pages 205–214, 1999.

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

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

[88] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets, 2001. Manuscript.

[89] B. Nguyen, S. Abiteboul, G. Cobena, and L. Mignet. Query subscription in an XML webhouse. In *Workshop on Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[90] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. ACM SIGMOD*, 2001.

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

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

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

[94] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Proc. ACM PODS*, pages 105–112, 1995.

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

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

[97] M. Spielmann. Verification of relational transducers for electronic commerce. In *Proc. ACM PODS*, pages 92–103, 2000.

[98] 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.

[99] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Saloma, editors, *Handbook of Formal Languages*. Springer-Verlag, 1997.

[100] M. Y. Vardi. The complexity of relational query languages. In *Proc. STOC*, pages 137–146, 1982.