Pergamon

Information Systems Vol. 23, No. 8, pp. 615–637, 1998

Printed in Great Britain
0306-4379/98 $19.00 + 0.00

FORMAL MODELS OF WEB QUERIES†

ALBERTO O. MENDELZON[1] and TOVA MILO[2]

[1]University of Toronto, Department of Computer Science, Toronto, Canada M5S 3H5
[2]Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel

(Received 20 February 1998; in final revised form 14 October 1998)

Abstract — We present a new formal model of query and computation on the Web. We focus on two important aspects that distinguish the access to Web data from the access to a standard database system: the navigational nature of the access and the lack of concurrency control. We show that these two issues have significant effects on the computability of queries. To illustrate the ideas and how they can be used in practice for designing appropriate Web query languages, we consider a particular query language, the *Web calculus,* an abstraction and extension of the practical Web query language *WebSQL.*

Key words: World Wide Web, Web Queries, Query Languages, Computability, Formal Models

1. INTRODUCTION

Tools and techniques for retrieving information from the World Wide Web are rapidly being developed [9, 10, 13, 4, 12, 8]. Most of these works are based on the metaphor of the Web as a database, in order to carry over and adapt familiar query languages such as SQL or Datalog. However, the Web is *not* a database, and querying it differs significantly from querying a conventional database. Two essential aspects that make a query on the Web different from a query on a standard database are the lack of concurrency control and the limited data access capabilities.

First, consider the impact of the lack of concurrency control on which queries can and cannot be effectively evaluated. Concurrency control on the Web is often very crude: the owner of a document can lock it to prevent others from accessing it while it is being modified, but no other user can do this, and there is no transaction mechanism. This makes many queries in principle impossible to compute in a finite time. For example, finding the documents reachable directly or indirectly from a given starting point may require following a chain in which documents are being added to the end of the chain faster than we can discover them, so the computation may never terminate.

Since the dynamic nature of the Web has such an effect on query computability, it is natural to ask what happens if we assume a static Web. For example, stronger concurrency control mechanisms in future versions of the Web might isolate a query from changes and provide a conceptually static view. What will be the effect of introducing such a mechanism on query evaluation? Even when updates to the Web can be ignored, many queries that would easily be computable on a conventional database are not so on the Web, because of the limitations in data access: access to Web documents is strictly navigational. There are only two ways to retrieve a document; either through its URL (uniform resource locator), or through some other document (possibly a form) that points to it.

This statement may seem false at first sight to anyone who has used an *index server* such as *Altavista* [7]. These are Web sites that provide associative access to a large collection of documents by using a searchable index (that is built off-line by navigation). However, from a theoretical point of view, these indices do not provide true associative retrieval from the whole Web – they do not guarantee they have indexed every document, nor do they give any guarantee of currency of the index. They are basically forms that provide access to some pre-computed data.

†Recommended by Gottfried Vossen

This limitation in data access has a strong impact on the feasibility of queries: a query such as "are there any documents pointing to document d?" is impossible to evaluate because, even in a static Web, there is no way to examine *all* documents and be sure that we have not missed any.

In this paper we propose a formal model of Web queries that highlights the effects that the limited access to data and the lack of concurrency control have on data retrieval. Our goal is to provide a notion of computable Web queries that will play the role that the computable queries of Chandra and Harel [6] have played in the theory of relational databases. Of course, there are other important aspects of the Web, such as heterogeneity, autonomy, and lack of structure, which we do not address here. Nevertheless, we believe that the model we propose can serve as a robust basis for the theoretical study of these and other issues.

To isolate the problems caused by the limited access to Web data from those caused by the lack of concurrency control, we start by analyzing the static case, that is, we ignore updates. We model the Web as a relational database containing a relation of *Node* objects, one per document, and a relation of *Link* objects, one per node-to-node link, plus possibly additional relations called *base relations*, containing auxiliary information, such as sets of initial URL's where the navigation may start. We present the data model in Section 2.

In Section 3 we present our notion of computable queries on the Web, which we call *Web queries*, as relational queries that can be computed by a special kind of Turing machine, called a *Web machine*. This machine has full access to the base relations, but can only access the Node and Link relations through an oracle that provides the navigation functions. We define a Web query as a relational query (in the sense of Chandra and Harel) that can be computed by a Web machine. Then we give two characterizations of Web queries, one in terms of the sub-Web they depend on, and one in terms of reflective relational machines [1].

Although both *Node* and *Link* relations are finite in our model, the access restrictions imply that certain queries can only be evaluated by a non-terminating computation. For example, the query "list all nodes" can be computed by enumerating all possible URL's and for each one testing whether it corresponds to a node; but there is no way to stop. Following Abiteboul and Vianu [2], we call such queries *eventually computable*, and in Section 4, we characterize them by a monotonicity property and by a new kind of reflective relational machine.

To illustrate the ideas and how they can be used in practice for designing appropriate Web query languages, we consider in Section 5 a particular query language, the *Web calculus,* an abstraction and extension of the practical Web query language *WebSQL* [13]. The calculus is essentially first-order logic plus regular expressions to specify paths between nodes. We show that, if not properly restricted, the full calculus can express queries that are not Web queries, such as "find all nodes with no incoming links". We then present the syntactic restrictions needed to express only Web queries, and various classes of eventually computable queries. In particular we define *safe* calculus formulas, and show that they express exactly the computable Web queries expressible in the calculus. We then relax this restriction to *semi-safe* queries, which express exactly the calculus queries that are eventually computable and have eventually computable complement.

Once the effect of the limited access to data is understood, we move in Section 6 to the issue of updates and lack of concurrency control. To account for the dynamic Web environment we now view a Web as an infinite sequence of *Node* and *Link* relation pairs, instead of a single pair. A *dynamic Web machine* is exactly like a Web machine, except that the oracle can, each time it is invoked, non-deterministically switch from one *Node, Link* pair to the next one in the sequence. We distinguish three classes of queries in the dynamic Web environment: First, *dynamic Web queries,* such as "list the titles of all documents directly reachable from d", whose computation terminates no matter how much the Web changes while computing. (This query can be evaluated by a program that reads d, traverses each link from d, fetches the target document, and prints its title. Note, however, that the answer may contain data from various states of the Web, and this has to be accounted for in the definition of queries.) The second class, *dynamic semi-computable Web queries*, consists of queries that *could* be evaluated by a terminating computation if the computation were sufficiently fast, that is, faster than the update rate. For example, "list all nodes reachable directly or indirectly from d" has this property. The third class is *eventually computable dynamic Web queries,* such as "list all documents in the Web", whose answers can be computed, but not by a terminating computation.

We then study the relationship between these classes and their static Web counterparts. The idea is that the semantics of Web query languages[13, 9, 10] is usually defined with respect to a static Web. However, in practice, when the program produced by the query compiler is executed, it runs in a dynamic environment where the Web keeps changing. In terms of Web machines, this means that the query is compiled into an ordinary (static) Web machine, but, at run time, the oracle in the machine is replaced by a dynamic one, and the dynamic machine is run instead. It is thus important to understand the relationship between the static and dynamic machines and the queries they compute. We present an exact characterization of this relationship.

Finally, in Section 7 we sketch how to extend the results to take into account Web *forms*, both in the static and dynamic cases. We show that, with appropriate assumptions, the results can be extended to handle forms and index servers. We conclude in Section 8.

Related Work

The only work we are aware of dealing with formal models of Web queries is Abiteboul and Vianu's [2], which is closely related to ours. There is however one central difference. According to Abiteboul and Vianu, "Perhaps the most fundamental aspect of our model is that we view the Web as infinite". The argument for this is that "exhaustive exploration is –or will soon become– prohibitively expensive". This is not entirely clear to us.

First, the Web is *finite* at any given moment, even if large. In fact, its size, measured in number of documents, is not enormous when compared to some very large databases. For example, the Altavista index server [7] has indexed, as of October 1998, about 125 million documents into an index of about 200 Gigabytes. The Altavista "crawler" can scan 6 million documents per day. Lawrence and Giles [11] estimate the size of the Web at about 320 million documents. By database standards, this is not very large. (For example, the Big Sur project at Berkeley [5], dealing with data from NASA's Earth Observing System, is designed for 1 Terabyte of data, or about 5 times the size of the Altavista index, and this is only 1/1000 of the ultimate data size that the EOS project will have to handle.)

Second, even if exhaustive exploration of the Web is or will become a practical impossibility, theoretical computer science draws a sharp distinction between intractable and impossible; problems whose solution is known to take longer than the predicted age of the universe are nevertheless considered decidable.

In our model, the Web is finite, but there is still non-terminating behavior, caused by two very different reasons which we treat separately: the restriction to navigational access, and the lack of concurrency control. This leads to a different classification of queries from the one of Abiteboul and Vianu, highlighting the specific effects of these two issues. For example, the query "find all documents reachable from document d" is computable in our static model, but not in theirs. In the dynamic Web, this query is only eventually computable, as it is in the Abiteboul and Vianu model. On the other hand, a query such as "list all documents" is eventually computable in both their model and ours (static and dynamic). And a query such as "does this node have any incoming link?" is not computable at all in any of the models.

2. PRELIMINARIES

We use a simple relational model of the Web to best highlight the essential computational issues. We first describe informally the main ideas, and then present our formal data model.

Documents and Links: The World Wide Web is a large, heterogeneous, distributed collection of documents connected by hypertext links. At the highest level of abstraction, it can be viewed as a graph whose nodes are Web objects that are identified by a Uniform Resource Locator (URL) and have some arbitrary content whose interpretation depends on its type (HTML, Postscript, image, audio, etc.). Also, Web servers provide some additional information such as the type, length, and the last modification date of an object. Moreover, an HTML document has a title and a text. So, for query purposes, we can associate a Web object with a tuple in a virtual Nodes relation:

$$N[id, title, content, type, length, modif, ...]$$

The *id* represents the URL and is thus a key, and all other attributes may be null.

HTML documents may contain hypertext links to (some offset within) other documents. A *hypertext link* is specified inside an HTML document by a sequence, known as an *anchor*, of the form `<A HREF=`*href*`>`*label*`</A>` where *href* (standing for *hypertext reference*) is the URL of the referenced document (plus perhaps an offset within the document), and *label* is a textual description of the link. An HTML document may contain the same anchor several times (at different offsets from the beginning).

We capture the information present in a link as a tuple in a Links relation:

$$L[source, destination, \ldots]$$

where *source* and *destination* are the Oid's of the origin and destination of the link. Additional attributes can describe link properties such as label†.

Forms: Forms are special Web documents allowing a user to fill in some parameters and get as a result another document. A form can therefore be viewed as a document whose outgoing links have parameters. The (virtual) documents pointed to by the parameterized links are the ones returned by the program associated with the form when called with the given arguments. Such parameterized links can be described in a relation L_p having the same structure as the links relation L with an additional attribute *param_set*. Taking a *nested values* approach, *param_set* could contain a set of tuples, one for each parameter name and values. To stay within the relational model, we can model this by having *param_set* contain some id for the parameter set, and using an auxiliary parameters relation

$$P[param\_set : t_1, param\_name : t_2, param\_val : t_3]$$

recording the relationship between the set id's and the actual parameter values. Note that index servers often have a form-based interface, thus they can also be modeled as nodes with outgoing parameterized links.

Base Relations and Data Access: People interacting with the Web often use additional data such as bookmarks, local files, and perhaps even some data stored in a database system. This data can be used to initiate a Web access, for example by following some selected bookmark. Abstractly, one can view all this additional information as a set of *base relations* containing the relevant data. The main difference between these relations and the N, L, L_p, and P relations is that the only way to access the Web relations is by explicitly specifying a URL, or following a link; while the base relations are fully accessible, say, using a relational query language. Note that, although index servers give the illusion of content search capabilities, they only provide an approximation to it. When an index server is queried, the query is typically evaluated on a pre-computed index, and the answer does not necessarily reflect the state of the Web at query time, or a complete and consistent state of the Web at any time.

Finiteness: The Web is very large, but at any given time, the set of documents in it, and the amount of data in them, are finite (although they keep changing and growing with time). Ordinary (non-form) HTML documents contain a finite number of outgoing links. For forms, this is not obvious, since they may have parameterized links. However, in practice, the number of possible combinations of parameter values that actually lead to documents is usually finite. (In particular, most parameter fields have a fixed size and thus can be fed only a bounded, but possibly very large, number of values.) We will thus assume in the sequel that the Node, Link, Parameterized-link, and Parameters relations N, L, L_p, P are finite at any given point in time. We shall deal with the changes over time to these relations separately.

†In fact we should also add an *offset* within the source document to allow for the possibility of the same link appearing several times in one document. We omit this for simplicity.

Some forms may accept input of arbitrary size and may generate infinitely many documents (e.g. a form that is an HTML editor). Analysis of such forms is beyond the scope of this paper.

We are now ready to present the formal data model. To simplify, we ignore for the moment the issue of forms and parameterized links and present a simplified version of the model. We shall come back to this issue in Section 7.

We assume an infinite set D of *data values*, and a finite set T of *simple types* whose domains are subsets of D. Tuple types $[a1 : t1, ..., a_n : t_n]$ with attributes a_i of simple type t_i, $i = 1 \dots n$ are defined in the standard way. The domain of a type t is denoted by $dom(t)$. We distinguish a simple type $Oid \in T$ of *object identifiers*, and two tuple types

$Node = [id : Oid, ..., a_i : t_i, ...]$

and

$Link = [source : Oid, \dots, destination : Oid, \dots, b_j : t_l, \dots]$

The domain $dom(Oid)$ is assumed to be infinite and recursively enumerable, and the attribute names in the two definitions are all distinct. We further assume that none of the attributes other than id, $source$, and $destination$ are of type Oid. We shall refer to tuples of the first type as *Node objects* and to tuples of the second type as *Link objects*. In our model of the World Wide Web, documents will be mapped to $Node$ objects and the hypertext links between them to $Link$ objects. In this context, the object identifiers (Oid) will be the URL's.

A *Web database schema* W is a relational database schema with a finite set of base relations $DB = \{R_1, \dots, R_n\}$, and two additional relation schemes, N containing node objects, and L containing link objects. An *instance* of a Web database schema W, also called a *Web database*, is a mapping associating a finite set of tuples with every relation name $R_i \in DB$, a *finite* set of node objects with N and a *finite* set of link objects with L, such that the following holds: (1) id is a key in relation N, and (2) there is a referential integrity constraint between the $source$ attribute in L and the id attribute in N.

Note that the referential constraint is only for the $source$ attribute and not for the $destination$. This is because documents can contain URL's pointing to non existing documents. When clear from the context, we shall overload notation and use $W = (DB, N, L)$ to denote both the Web database schema and the instance.

3. QUERIES

We now formalize the notion of queries on Web databases. The standard definition requires queries to be generic and computable by a Turing machine on some encoding of the database. While in ordinary databases the user may have full control of the database relations, and thus a standard Turing machine is an adequate abstraction of the computation power, in the Web context the access to data is limited to navigation. (Again, recall that index servers and similar search engines can be viewed as navigational access that starts from a form and follows parameterized links to pre-computed data.)

This means that computing on the Web is not the same as computing on an ordinary database containing the Web data. For example, suppose the Nodes and Links are stored in an ordinary database. With a standard query language, one could easily list the titles of all the documents. This simple query is, however, impossible to answer with the current Web architecture, because the *Nodes* relation is never fully available to the user. The user may attempt to compute the query by starting from some given set of URLs and list everything reachable from them; but this will not reach disconnected portions of the network. Even using index servers may not help because they too may have failed to discover some disconnected regions. One way to try and discover these unknown parts of the network is to start enumerating all the possible URL's and check for each one whether it points to a real document. But assuming the domain of URL's is infinite, there is no way to know when all documents have been covered and stop the enumeration.

We want our definition of Web queries to highlight these differences from standard database computation, and to capture the limited computation power. We thus refine the standard definition of query. The idea is to require that a query not just be computable by an ordinary Turing machine, but by a more limited machine simulating the computation power available on the Web. We introduce *Web machines* to formalize this limitation.

Definition 1 A *Web machine* is a Turing machine augmented with an oracle. It has two input tapes, an ordinary tape on which the input DB base relations are encoded, and an oracle tape on which the N and L relations are encoded. It also has two working tapes, an ordinary working tape and an oracle working tape. Finally, it has an output tape on which the result of the computation is written in an *append only* manner[†].

At each point of the computation, the machine can do one of the following:

1. read from the ordinary input tape or from any of the working tapes (it cannot read from the oracle input tape),
2. write on the output or any of the working tapes,
3. call the oracle.

In the last case, the oracle reads the content of its working tape, interprets its content as a node id, and copies (from its input tape) to the working tape the relevant (encoded) node tuple, if a node with such an id exists, and also the tuples of all its outgoing links. If no such node exists, the oracle simply writes a special symbol $\perp$ on the oracle's working tape.

The machine captures the idea that nodes can be accessed only in a limited way – by explicitly specifying their id. To navigate starting from a given URL, the oracle is first called for this URL. It returns, among other things, the id's of documents pointed to by this document. Then it can be called again on these id's to access their data, and so on.

Definition 2 A *total relational query* is a generic[‡] mapping Q from instances of a database schema to instances of another one, such that there exists a Turing machine that, given any instance I of the first schema and any encoding $enc(I)$ on its input tape, the machine halts with $enc(Q(I))$ on its output tape.

Note that this is more restrictive than the traditional definition of *relational queries* in [6], that allows partial recursive functions. For brevity, we omit the word *total* in the rest of this paper, and whenever we say relational query we mean a total one.

Definition 3 A *Web query* is a relational query Q mapping Web database instances to sets of tuples over the values in the instances, such that there exists a Web machine that computes Q on any Web. That is, given any Web database $W = (DB, N, L)$, and any encoding enc, when started with $enc(DB)$ on its ordinary input tape and $enc(N)$, $enc(L)$ on its oracle input tape, the machine halts with output $enc(Q(W))$ on its output tape.

The following are examples of Web queries:

- List the title of nodes reachable from the node with oid o by a path of length less than 3.
- Find all nodes on a cycle of length at most 3 containing the node with oid o.
- Find the nodes reachable from the node with oid o.

Note that the last query above is a Web query by our definition but is not computable according to [2] because they assume that the Web is infinite.

We next characterize Web queries. From the characterization (in Theorem 1 below) it will follow that queries like the following are not computable in the Web context (even though they are computable in the standard database context).

[†]The use of multiple tapes and the *append only* manner of the output tape is insignificant. It is used only to simplify the presentation. A similar definition can be given for a single ordinary tape.

[‡]A *generic* mapping is one that commutes with domain isomorphisms [6, 15].

- Find all nodes.
- Find all nodes referencing the node with oid o.
- Find all nodes with no incoming links.

Before presenting the characterization, we need some auxiliary notations and definitions.

Definition 4 Given some Web database instance W and a set of values S. The *reachability set* of S w.r.t W, denoted S^*_W, is the union over all $i \geq 0$ of S^i_W, defined inductively as follows:
(1) $S^0_W = S$
(2) $S^{i+1}_W = S^i_W \cup \{v \mid v$ is an attribute of some node object in W with id in S^i_W, or of some of its outgoing link objects $\}$

Since Web instances are finite, so is S^*_W for any finite set S. Basically, S^*_W describes all the data reachable starting from nodes with id in S and following links.

We now define an equivalence relationship between Web instances, based on the reachability set, and then use it for the characterization. For a relation R and a set of values S^*_W, we use $R{\downarrow_{S^*_W}}$ to denote R when restricted to the tuples where all attribute values are in S^*_W.

Definition 5 Given a set S of values, the equivalence relation $=_S$ among Web instances is defined as follows. $W =_S W'$, for two Web instances $W = (DB, N, L)$,$W' = (DB', N', L')$ over the same schema, iff (1) $DB = DB'$, (2) $N{\downarrow_{S^*_W}} = N'{\downarrow_{S^*_{W'}}}$, and (3) $L{\downarrow_{S^*_W}} = L'{\downarrow_{S^*_{W'}}}$.

Informally this means that the two Webs contain the same base relations and the same sub-network induced by S and the nodes reachable from S. In particular, the nodes of this sub-network have the same properties in both graphs, and in both graphs this sub-network points to the rest of the world in the same way.

We are now ready to present the characterization.

Theorem 1 *A relational query Q, mapping a Web database to a set of tuples over the values in the Web database, is a Web query iff for every two inputs $W = (DB, N, L)$, and $W' = (DB, N', L')$ such that $W =_S W'$, where S is the set of values mentioned in DB, it is the case that $Q(W) = Q(W')$.*

Proof. *If:* Assume that Q is a relational query with the property that for every two inputs $W = (DB, N, L)$ and $W' = (DB, N', L')$, such that $W =_S W'$, where S is the set of values mentioned in DB, it is the case that $Q(W) = Q(W')$. We have to show that there is a Web machine that computes $Q(W)$ for every instance W.

Now, consider an instance $W = (DB, N, L)$, and let $W' = (DB, N{\downarrow_{S^*_W}}, L{\downarrow_{S^*_W}})$, where S is the set of values mentioned in DB. Clearly $W =_S W'$, hence $Q(W) = Q(W')$. Thus to compute the result of Q on W it suffices to be able to compute it for W'. The Web-machine for Q can thus be defined as follows: It works in two phases. In the first phase the oracle is called repeatedly to retrieve all the nodes and links of W' and write them on the working tape. (This is possible because W' contains exactly all the nodes and links reachable from ids in DB). Then, the ordinary Turing machine for Q will be run on the retrieved W'. (Such a machine exists because Q is assumed to be a relational query).
Only if: Let $W = (DB, N, L)$ be some Web-instance, let S be the set of values in DB. Consider $W' = (DB, N{\downarrow_{S^*_W}}, L{\downarrow_{S^*_W}})$. To prove this direction it suffices to show that for every instance W'' s.t. $W' =_S W''$ it is the case that $Q(W') = Q(W'')$. (Since $W =_S W'$, this in particular will prove that $Q(W) = Q(W'')$ for all such W''.)

We look at the computation of a Web machine for Q when operating on W'. Since the computation terminates, the number of calls to the oracle is finite. Let O be the set of node ids on which the oracle is called.

Now consider some $W'' =_S W'$. W'' contains the subgraph reachable from ids in S plus perhaps some other nodes and links. (Note however that since $W'' =_S W'$, none of these additional nodes is reachable from nodes with id in S.)

Case 1: Assume first that the ids of all these additional nodes are disjoint from the ids in O, and consider the computation of the Web-machine when working on W''. We claim that this computation must be identical to the computation of the machine when working on W'. The proof is by induction on the number of oracle calls in the computation. That is, the inductive hypothesis is: the two computations are identical up to right before the nth oracle call. The basis, $n = 1$, is clear since W'' and W' agree on DB. Now suppose the computation of the machine on W'' is identical to the computation on W' up to right before the $n-1$st oracle call, for $n > 1$. By the inductive hypothesis, the oracle tape right before the $n-1$st oracle call has exactly the same content in the two runs. So the oracle is called in the two runs on the same id o, hence o must be in O. We argue that the oracle returns the same answer in the two runs. First, if o is reachable from S, then, since W'' agrees with W' on S, the node with id o exists in W'' and has the same structure and the same outgoing links as in W'. Otherwise, if o is not is reachable from S, then, since W' contains only nodes reachable from S, the oracle returns $\perp$ when called on W'. Since all the ids of the additional nodes in W'' are disjoint from the id's in O, then W'' does not have a node with id o either, so the oracle will return $\perp$ here as well. So, right after the $n-1$st oracle call the machines are in the same configuration, hence their computation will be the same until right before the nth oracle call, or termination, whichever comes first.

Case 2: The other possibility is that some of the additional ids in W' belong to O. For this case, we first show that W'' has an isomorphic instance with no additional ids in O:

The isomorphic instance looks exactly like W'' except that all the ids not in S^*_W are replaced by some new ids distinct from the ids in O. Note that it is always possible to find such an isomorphic instance because (1) the domain of oids is infinite, and (2) no attributes other than *id* (in N) and *source*, *destination* (in L) are of type *Oid*.

Now, let H be the isomorphism mapping and let $H(W'')$ be the isomorphic Web. Note that the isomorphism maps nodes in S^*_W to themselves. Also note that W' contains only nodes in S^*_W, thus all the values in $Q(W')$ are from S^*_W. Since Q is assumed to be a relational query, it is generic, and we have that $Q(H(W'')) = H(Q(W''))$. From Case 1 above, $Q(H(W'')) = Q(W')$. Hence, $H(Q(W'')) = Q(W')$. Since $Q(W')$ contains only nodes in S^*_W, and since the only nodes mapped by H to such nodes are exactly the same nodes in S^*_W, for this equivalence to hold it must be the case that $Q(W'') = Q(W')$. □

Basically this means that Web queries are only interested in the sub-Web reachable starting from data in the base relations and following links, and ignore all other nodes. In particular, the first part of the proof implies that a Web query can always be computed in two phases: First all the reachable documents are retrieved, and then the query is evaluated on them (and the base relations) without any further access to the Web.

Remark: To deal with the use of constants in queries, one needs to talk about C-generic queries rather than just generic queries. A query is C-generic with respect to a finite set of constants C, if it commutes with all domain isomorphisms that leave C fixed. We can naturally extend our definition and talk about C-generic Web queries – these are simply C-generic relational queries that are computable by a Web machine. C-genericity allows the use of constants, so in particular constant URLs can be used in queries. Once this is allowed, the above theorem becomes:

Theorem 2 *(Theorem 1 - revisited) A C-generic relational query Q, mapping a Web database to a set of tuples over the values in the Web database, is a C-generic Web query iff for every two inputs $W = (DB, N, L)$, and $W' = (DB, N', L')$ such that $W =_S W'$, where S is the set of oid's values in DB or C, it is the case that $Q(W) = Q(W')$.*

The proof works exactly as above, except that now S contains also the values in C. □

The following is an immediate consequence of the above theorem. We say that a Boolean Web query Q is trivial if either for every instance I $Q(I) = \emptyset$, or for every instance I $Q(I) = \{[\,]\}$.

Corollary 3 *When there is no input database and no constants, Web queries are trivial.*

The above theorem characterizes Web queries as a special sub-class of relational queries. Is it possible to give the Web queries a direct characterization independent of relational queries?

For standard relational queries, such characterizations exist in the literature, and are given in terms of languages [6] or machines [3] that compute exactly all the relational queries. It turns out that a similar characterization can be given for Web queries as well. For that we use *Web relational machines*, a slight refinement of Abiteboul and Vianu's *reflective relational machines* [1]. We next briefly recall the definition of such machines (for a full definition and study of properties see [1]), and then define Web relational machines.

Definition 6 A *reflective relational machine* is a Turing machine with a special query tape and a separate *relational store*, capable of storing arbitrary relations. Initially, the relational store contains only the input relation R_0. The machine computes as an ordinary Turing machine, except that whenever the machine enters a special "query state", the contents of the query tape are interpreted as a first-order query, which is evaluated in a single step. The results of the evaluation are either stored in another relation in the relational store or, if the query was a Boolean query, communicated to the machine on the query tape.

Definition 7 A *Web relational machine* is a variant of the reflective relational machine. It gets a Web database as input[†] and works like an ordinary reflective relational machine, except that the only two queries it can use to access the relations N and L are: $\{ n_1, ..., n_i \mid R(n_1) \wedge N(n_1, ..., n_i) \}$ and $\{ l_1, ...l_j \mid R(l_1) \wedge L(l_1, ..., l_j) \}$, where R is some fixed relation name different from N, L.

It was shown by Abiteboul, Papadimitriou and Vianu [1] that reflective relational machines compute exactly all the relational queries. The following theorem shows a similar relation between Web relational machines and Web queries.

Theorem 4 *Web relational machines compute exactly all the Web queries.*

Proof. *Direction 1:* The simulation of a computation of a Web relational machine on a Web machine works as follows. All computations/queries not involving N and L are simulated on the tape using the encoded base relations. The access to N and L is simulated by iterating on all the values in (the encoded version of) R, calling the oracle for each of them, and using the relevant part of its answer.

Direction 2: Given a Web query, we construct a Web relational machine for it as follows. By Theorem 2, it suffices to first construct the relations $N \downarrow_{S^*_W}, L \downarrow_{S^*_W}$ (where S is the set of values in DB and C) and then run a relational query on them. The machine first interacts with the N, L relations using queries of the above form, and, by iteratively applying the queries, builds the relations $N \downarrow_{S^*_W}, L \downarrow_{S^*_W}$: At the first iteration, the relation R is built so that it contains all the values in DB and C. Then at each iteration R contains the union of this set plus the values of all the attributes in the relations computed so far.

Once this is done we are back to the case where we have a standard database and a generic computable query to compute on it, and it was shown in [1] that all such computations can be simulated. □

4. EVENTUALLY COMPUTABLE QUERIES

There are many relational queries that are not Web queries. Examples of such queries are "list all documents referencing o", or "list the titles of all articles". The termination of computation of such queries cannot be guaranteed. Nevertheless, it is possible to effectively produce all the tuples in their output (though we may never know that we are done). For example, the query "list all

[†]Note that reflective machines have only one input relation; this is not an important difference, since we can always take the Cartesian product of all input relations, then use projection.

documents referencing o" can be computed by enumerating all possible URL's and for each one testing whether it corresponds to an existing node, and whether this node points to o; but there is no way to stop. Following [2] we call such queries *eventually computable.*

To formally define this, we first explain what it means for a relational query Q to be *eventually computed* by a Web machine.

Definition 8 Let Q be a mapping from instances of a Web database to relations. We say that Q is *eventually computable* by a Web machine if there is a Web machine whose computation on any Web database encoded as $enc(W)$ has the following properties:

1. the contents of the output tape at each step of the computation form a prefix of $enc(Q(W))$, and

2. for each tuple in $Q(W)$, its encoding appears on the output tape after a finite number of steps in the computation.

The notion is the same as in [2], except that our Web machine is different from theirs. We now use it to define the *eventually computable* Web queries.

Definition 9 An *eventually computable* Web query (EC Web query, for short) is a relational query (in the sense of Definition 2) mapping Web database instances to sets of tuples over the values in the instances, that is eventually computable by a Web machine.

The notion is again similar to that of Abiteboul and Vianu [2], but there is one significant difference - our definition of EC Web queries requires Q to be a relational query, not an arbitrary partial function. We argue that this is essential if we want to distinguish between non-computability caused by the limited access capabilities of the Web, and non-computability that has only to do with the complexity of the function being evaluated (independently of whether the input is ordinary relational data or Web data).

The two queries above, "list all documents referencing o", and "list the titles of all articles", are examples of EC Web queries. On the other hand, we shall show below that the query "Find all nodes with no incoming links" is not an EC Web query. Intuitively this is because there is no way to ensure that we have examined *all* nodes to conclude that some node has no incoming links. Formally, this will follow from Theorem 5 below.

As for Web queries, we give a characterization of the sub-class of the relational queries that are EC Web queries. The characterization is based on a monotonicity property: at each point of the computation, the machine has seen only a part of the Web and does not know what the rest of the Web looks like. Thus if a tuple is output at some point, it will also be output whenever the query is evaluated in a network "containing" the sub-network seen so far. This notion of containment has to be defined with some care.

Definition 10 Let S be a set of values. Define a class $\mathcal{W}_S$ of Web instances, and a partial order $\leq_S$ on them, as follows: (1) An instance that contains nodes for all the oid's in S (plus perhaps some more nodes), is in the class. (2) If W, W' are in the class, and W' is obtained from W by adding a set of nodes N', and a set of outgoing links L' for them, such that the nodes in N' are unreachable (in the extended instance) from nodes in W, then $W \leq_S W'$.

Theorem 5 *A (C-generic) relational query Q, mapping a Web database to a set of tuples over the values in the Web database, is an EC Web query iff for every input $W = (DB, N, L)$, Q is monotone with respect to $\leq_S$, where S contains all the values mentioned in DB (or C).*

Before going into the proof, observe that the theorem implies that every EC Web query Q can be computed in a monotone manner as follows:

- First evaluate Q with respect to the sub-Web reachable using values in DB (and C), and output the result. Then,

- start enumerating oids, and whenever finding a new node (1) recompute Q with respect to the sub-Web reachable using this oid plus all values in the sub-Web of the previous step, and (2) add to the output all the new tuples.

Note that the sub-Webs computed in the iterations obey the $\leq_S$ order, and that eventually we will enumerate all the nodes in the given Web. Theorem 5 assures that if a tuple is output in one of the iterations it will also be in the output of later iterations and, in particular, is still in the output when the whole Web has been seen. The fact that Q is a relational query means that it is computable in the traditional sense and thus each iteration step will halt.

We are now ready to prove the theorem.

Proof. *If:* Assume that Q is a relational query that is monotone with respect to $\leq_S$, where S contains all the values mentioned in DB (or C). The fact that Q is an EC Web query follows from the computation described above, which can be easily done using a Web Machine: The reachability set of the values in DB and C can be computed using the oracle, and similarly for the reachability set of each new id in the enumeration. Once the relevant nodes and links are written on the working tape, the query Q can be evaluated at each step by running the regular Turing machine of Q. (Note that since Q is a relational query such a machine exists and is guaranteed to terminate).
Only if: We need to show that all the EC Web queries Q are monotone with respect to $\leq_S$, where S contains all the values mentioned in DB (or C), i.e. that for every two Web instances W', W'', if $W' \leq_S W''$ then $Q(W') \subseteq Q(W'')$.

The proof is very similar to that of Theorem 1. We look at the computation of the Web machine for Q when operating on W', and consider the point in time where all the tuples in the results have already been generated. (Since the query is EC, such a finite point in time must exist.) The number of calls to the oracles up to this point is finite. Let O be the set of node ids on which the oracle was called in this run. Now let W'' be some Web instance s.t. $W' \leq_S W''$. By the definition of $\leq_S$, W'' contains all the nodes in W' and perhaps some additional nodes. As in the proof of Theorem 1 we consider two cases: Case 1 where all ids of the additional nodes of W'' are distinct from the ids in O, and Case 2 where some of the additional ids belong to O.

For Case 1, we can use the same induction as in the proof of Theorem 1 to show that the computation of the Web machine when operating on W'' must be identical up to this point, and thus outputs the same tuples. (Additional tuples may be output afterwards.) Hence $Q(W') \subseteq Q(W'')$.

For Case 2 we can use a similar genericity argument as in case 2 in the proof of Theorem 11. We first show that W'' has an isomorphic instance with no additional ids in O: The isomorphic instance looks exactly like W'' except that all the ids not in W' are replaced by some new ids distinct from the ids in O[†]. Now, let H be the isomorphism mapping and let $H(W'')$ be the isomorphic Web. Since Q is assumed to be a relational query, it is generic, and we have that $Q(H(W'')) = H(Q(W''))$. From Case 1 above, $Q(W') \subseteq Q(H(W''))$. Hence, $Q(W') \subseteq H(Q(W''))$. Since $Q(W')$ contains only nodes in W', and since the only nodes mapped by H to such nodes are exactly the same nodes in W', for this equivalence to hold it must be the case that $Q(W') \subseteq Q(W'')$. □

As for Web queries, an independent characterization for EC Web queries also exists, and can be given in terms of a special variant of reflective relational machines.

Definition 11 A *monotone relational machine* is a reflective relational machine that takes as input a Web database $W = (DB, N, L)$ and works in two phases.

Phase 1: For all subsets S_i of values from W including all the values in DB, compute the sub-Web N_i, L_i of N, L, containing all the nodes reachable from nodes with id in S_i, and all their outgoing links.

Phase 2: Run some reflective relational machine on all the instances $W_i = (DB, N_i, L_i)$ (the same machine on all the instances), and output the union of the results.

[†]Here again, note that it is always possible to find such an isomorphic instance because (1) the domain of oids is infinite, and (2) no attributes other than *id* (in N) and *source*, *destination* (in L) are of type *Oid*.

Theorem 6 *Monotone relational machines compute exactly all the EC Web queries.*

Proof. *Direction 1:* Every query Q computable by a monotone relational machine is clearly generic (and of course also computable by a Turing machine). Hence it is a relational query. Also, by the definition of the machine, the query is also monotone w.r.t $\leq_S$, S being the set of values in DB. Hence, by Theorem 5 Q is an EC query.
Direction 2: Consider an EC Web query Q. By definition it is also a relational query, hence by [1] there is a reflective relational machine RQ for it; the monotone reflective machine for Q runs this RQ in parallel in phase 2. Because of the monotonicity of Q, all the tuples computed in this phase for the sub-Webs should be indeed in the result. Hence the constructed machine indeed computes Q. □

It is possible to enumerate the tuples in the output of EC Web queries. Database theory often deals with the membership problem – testing whether a given tuple is in the result – rather than in computing the whole result.

Definition 12 For a relational query Q, its *membership query* M_Q is a Boolean query on the same schema as Q, augmented with an extra relation R_0, of the same arity as the result of Q, such that for any database instance I, $M_Q(I)$ is true if and only if $R_0(I) \subset Q(I)$, where $R_0(I)$ is the value of relation R_0 in instance I.

Clearly, if Q is an EC Web query, then its membership query is also an EC Web query. But in some cases it is also a Web query, i.e. computable by a halting Web machine.

The following is a rather standard fact in computability theory.

Proposition 7 *Let Q be a (C-generic) relational query, mapping a Web database to a set of tuples over the values in the Web database. The membership query for Q is a Web query iff both Q and its complement are EC Web queries.*

Proof. *Only if:* If the membership query for Q is a Web query, then both Q and its complement can be computed as follows: We first construct all the possible tuples with values in DB, test membership for each such tuple t (i.e. compute M_Q with $R_0 = \{t\}$), and output the tuples for which the result is positive (for Q) or negative (for the complement of Q). Then we start enumerating oids. For each oid we call the oracle. If a new node is found then we construct all the new tuples t with values from this new node, its links, and values in the DB, and again test membership for the tuples, outputting the ones for which the test is positive (for Q) or negative (for the complement of Q).
If: If both Q and its complement are EC Web queries then the membership can be computed as follows: We run in parallel the two machines until each of the tuples in R_0 is output by one of the machines. (Note that since Q and its complement are both EC, each tuple must be output on one of the output tapes in a finite time). We return a positive answer if all the tuples in R_0 appear on the output tape of Q's machine, and negative otherwise. □

Together with Corollary 3, this implies the following.

Corollary 8 *Let Q be a generic Boolean query on a schema where DB is empty. If both Q and its complement are EC Web queries, then Q is trivial.*

5. A WEB CALCULUS

In this section we study a particular query language in light of the various classes of queries described in the previous sections. Our language, which we call the *Web calculus*, is an extension and an abstraction of an implemented language, *WebSQL*, described in [13]. WebSQL integrates content-based retrieval, as provided by index servers on the Web, with structure and topology-based retrieval. For example, suppose we want to find documents with a title containing the string "database" that can be reached from the Computer Science Department's home page through paths of length 2 or less, without leaving the local server. The following query expresses this.

SELECT	*d.url*, *d.title*
FROM	Document d SUCH THAT
	"www.cs.toronto.edu"$(= \mid \rightarrow \mid \rightarrow\rightarrow)$ d
WHERE	*d.title* CONTAINS "database";

The regular expression $(= \mid \rightarrow \mid \rightarrow\rightarrow)$ restricts the path starting at the URL "www.cs.toronto.edu" to have zero, one, or two "local" links, i.e., links pointing to documents on the same server. The semantics of WebSQL is given using *Web calculus.* This is essentially first-order logic, enriched with path expressions. The vocabulary includes predicates for each base relation $\{R_1, \ldots, R_n\}$, node and link predicates N, L for the Web relations, and a $Path$ predicate that asserts the existence of a path of links between two Web nodes (and gives the full language higher expressive power than first-order).

The Web calculus for a Web database schema (DB, N, L) is the set of first-order formulas with equality on the following vocabulary:

- A predicate symbol R_i for each base relation in DB, with the same arity as the relation;
- Predicate symbols N and L, with the same arity as the corresponding relations;
- A ternary predicate $Path(n_1, R, n_2)$; the semantics is that n_1 and n_2 are nodes from the nodes relation N and there exists a path between them using links in the links relation L. Furthermore, R is a regular expression on an alphabet of link types, and the links along the path are restricted to satisfy R. We will not be making use of the regular expression in this section; see [13] for details.
- A binary *containment predicate* that takes as first argument an oid and as a second argument an atomic value; it means that the string represented by the second argument occurs within the body of the document represented by the first argument. We write it in infix notation as n `contains` `s`.

However, we need to restrict the syntax so that only computable or eventually-computable queries are expressible. For example, in the full language a query such as "find nodes with no incoming links" is easily expressible as

$$\{x \mid N(x, \ldots) \wedge \forall y (N(y, \ldots) \rightarrow \neg L(y, x, \ldots))\}$$

but we know this query is not eventually computable by Theorem 5. In this Section, we give a syntactic restriction of the Web calculus, called the *safe* Web calculus, and show that it captures exactly the Web calculus queries that are Web-computable; we also give a weaker restriction that captures exactly the queries Q such that both Q and its complement are eventually computable.

Two kinds of restrictions are needed to weed out formulas that express non-computable Web queries. First, we have to ensure that the first argument in a N, L, or $Path$ atom is bound to a known set of nodes; this is similar to what is called *groundedness* in [13], and *source-safety* in [2]. Second, as in the usual relational calculus, we have to ensure that arguments of a negated atom are instantiated and that terms of a disjunction use the same set of variables. The restrictions are specified as follows.

Definition 13 A formula in the *safe* Web calculus is a formula in the Web calculus which is of one of the following forms. In the list below, a is always a constant, $x, x_1, \ldots, x_n$, $y, y_1, \ldots, y_m$ are variables or constants, and $\phi(x_1, \ldots, x_n)$ and $\phi'(x_1, \ldots, x_n)$ are safe formulas whose free variables are exactly $x_1, \ldots, x_n$.

- $N(a, x_1, \ldots, x_n)$, $L(a, x_1, \ldots, x_n)$, $Path(a, R, x)$,
  $R_i(x_1, \ldots, x_n)$, $x = a$

- $(\phi(x_1,\ldots,x_n) \vee \phi'(x_1,\ldots,x_n))$,
  $(\phi(x_1,\ldots,x_n) \wedge \phi'(y_1,\ldots,y_m))$,
  $(\phi(x_1,\ldots,x_n) \wedge \neg\phi'(x_{i_1},\ldots,,x_{i_j}))$ (where the x_{i_j}'s are taken from among $\{x_1,\ldots,x_n\}$),
- $(\phi(x_1,\ldots,x_n) \wedge x_i = x_j)$,
  $(\phi(x_1,\ldots,x_n) \wedge x_i \neq x_j)$,
  $(\phi(x_1,\ldots,x_n) \wedge x_i = y)$,
  $(\phi(x_1,\ldots,x_n) \wedge x_i$ `contains` $\mathtt{x_j})$,
  $(\phi(x_1,\ldots,x_n) \wedge x_i$ `contains a`$)$,
  $(\phi(x_1,\ldots,x_n) \wedge L(x_i,y_1,\ldots,y_m))$,
  $(\phi(x_1,\ldots,x_n) \wedge N(x_i,y_1,\ldots,y_m))$,
  $(\phi(x_1,\ldots,x_n) \wedge Path(x_i,R,y))$
- $\exists x_i \phi(x_1,\ldots,x_n)$

The next theorem shows that safe formulas capture all the Web queries expressible in the Web calculus.

Theorem 9 *A Web calculus formula expresses a Web query if and only if it is equivalent to a safe Web calculus formula.*

Proof. *If:* It is easy to see that every safe Web calculus query can be computed by a Web machine that halts on all inputs. The proof is straightforward, by structural induction on the formula, and is thus omitted. It immediately follows that every Web calculus query that is equivalent to such a safe query can be computed by the machine of the equivalent safe query.
Only if: To prove that every Web query expressed by the calculus can be formulated in a safe manner, we use Theorem 2. The idea is that one can construct a safe formula computing the reachability set of values in DB (and constants C in the query), and then use this formula to rewrite the original calculus query ϕ so that all its node variables range only over this domain.

We first show the formula RS for computing the reachability set and then explain how it is used.

The formula for RS is defined using some auxiliary formulas: For a DB with relations $R_1,\ldots,R_m$ of arity $n_1,\ldots n_m$ resp., and a set of constants C, the formula

$$\rho(x) = (\bigvee_{i=1\ldots m, j=1\ldots n_i} \exists x_1 \ldots x_{n_i}\ R_i(x_1,\ldots,x_{n_i}) \wedge x = x_j\ ) \ \vee\ (\bigvee_{c\in C} x = c\ )$$

is satisfied by the values appearing in DB and C. The sets of node and link objects reachable from ids in this set are defined, resp., by the following two formulas.

$$\nu(x\ldots) = \exists y\ \rho(y) \wedge Path(y,*,x) \wedge N(x,\ldots)$$

$$\lambda(x\ldots) = \exists y\ \rho(y) \wedge Path(y,*,x)) \wedge L(x,\ldots)$$

Finally, the reachability set consisting of all values appearing in tuples that satisfy ρ, ν, and λ is defined by

$$RS(x) = \rho(x) \vee (\bigvee_{i=1\ldots p} \exists x_1 \ldots x_p\ \nu(x_1,\ldots,x_p) \wedge x = x_i) \vee (\bigvee_{i=1\ldots q} \exists x_1 \ldots x_q\ \lambda(x_1,\ldots,x_q) \wedge x = x_i)$$

Now we can transform the calculus query ϕ into an equivalent safe formula $\phi\prime$ by essentially conjoining RS atoms to ϕ to make sure that the resulting formula is safe: First we use standard techniques to remove the $\vee$ and $\forall$ symbols from the formula, if they exist. Then, we proceed by induction on the structure of ϕ. In the following the x_i's can be variables or constant symbols.

- If ϕ is an atom of the form $R_i(x_1,\ldots,x_n)$ for a relation R_i in DB, then ϕ is safe, there is nothing to do.

- If ϕ is an atom of any other form, that is, $N(x_1, \ldots, x_n)$, $L(x_1, \ldots, x_m)$, $x_i = x_j$, $x_i \neq x_j$, $x_i \texttt{contains x}_\texttt{j}$, $Path(x_i, R, x_j)$, then the conjunction of ϕ and all the formulas $RS(x_i)$ such that x_i is a variable in ϕ is safe and equivalent to ϕ by Theorem 2.

- If ϕ is of the form $\phi_1(x_1, \ldots, x_n) \wedge \phi_1(y_1, \ldots, y_n)$, then recursively transform ϕ_1 and ϕ_2 to equivalent safe formulas to obtain a safe formula equivalent to ϕ.

- If ϕ is of the form $\exists x_i \phi_1(x_1, \ldots, x_n)$, then recursively transform ϕ_1 to a safe formula as above.

- If ϕ is of the form $\neg\phi_1(x_1, \ldots, x_n)$, then recursively transform ϕ_1 into a safe formula $\phi_1\prime$ as above, and take the conjunction of (1) $\neg\phi_1\prime$ and (2) the conjunction of all the $RS(x_i)$ formulas for the free variables x_i in $\phi_1\prime$. □

Clearly, there are EC Web queries, such as "list all the nodes in the Web," that are not expressible as safe calculus formulas. We thus define *weakly safe* formulas, that express a large set (but not all) of the EC Web queries expressible in the Web calculus. These are defined using *semi safe* formulas that capture exactly those queries expressible in the Web calculus such that both the query and its complement are EC.

Definition 14 A Web calculus formula $\phi(x_1, \ldots, x_n)$ is *semi-safe* if the formula $\phi(a_1, \ldots, a_n)$ is safe, where the a_i's are constants.

For example, the query above, "list all the nodes", can be expressed by the semi-safe formula $N(x, \ldots)$. It is not hard to see that semi-safe formulas express only eventually computable Web-queries. However, there are eventually computable Web-queries in the calculus that are not expressible in this way. For example: find all nodes with some incoming links. This is expressible as

$$N(x, \ldots) \wedge \exists y N(y) \wedge L(y, x, \ldots))$$

which is not semi-safe. The fact that this query is not expressible by a semi-safe formula follows from the next theorem.

Theorem 10 *A Web calculus formula ϕ is equivalent to a semi-safe formula if and only if both the query expressed by ϕ and its complement are EC Web queries.*

Proof. For the (only if) direction, note that a semi-safe formula $\phi(x_1, \ldots, x_n)$ and its complement can both be eventually computed by a Web machine that enumerates all the instantiations $a_1, \ldots, a_n$ for the variables $x_1, \ldots, x_n$ and for each one evaluates the Boolean Web query expressed by $\phi(a_1, \ldots, a_n)$.

For the (if), note that by Proposition 7 it suffices to show that every Web calculus query whose membership testing is a Web query is expressible by a semi-safe formula. Suppose there is a Boolean Web query M that tests, given a tuple $x_1, \ldots, x_n$ in relation R_0, whether $\phi(x_1, \ldots, x_n)$ holds in a Web instance. By Theorem 9, there is a safe formula ϕ_M that expresses M. Note that ϕ_M has no free variables. Now let $\phi\prime_M(x_1, \ldots, x_n)$ be the formula ϕ_M where each occurrence of an atom $R_0(z_1, \ldots, z_n)$, where R_0 is the special relation symbol for which the membership is tested and the z_i's are variables or constants and, has been replaced by the conjunction of the equalities $x_1 = z_1, \ldots, x_n = z_n$, and the x_i's are variables not occurring in ϕ_M. Then $\phi\prime_M$ is a semi-safe formula that computes ϕ. To see this use simple induction on the structure of the formula. (Omitted). □

Note that, together with Proposition 7, this theorem yields that a Web calculus formula that expresses query Q is equivalent to a semi-safe formula if and only if the membership query for Q is a Web query. It is open whether there are syntactic restrictions that capture exactly all the eventually computable Web calculus queries.

Finally, we note that all the queries expressible in the WebSQL language of [13] are EC Web queries, and those satisfying the "groundedness" restriction in that paper are Web queries.

6. DYNAMIC ENVIRONMENT

The Web keeps changing and growing. In the current architecture no global concurrency control mechanism exists and thus the computation of queries is affected by the dynamic environment.

One issue is that a query that accesses the same document several times may see different things because the document was updated between the accesses. To some extent this can be avoided by using local document storage – whenever a document is read it is stored on the local disk, and further access to the document can be directed to this storage. This is a common practice in many Web tools.

A computationally more significant issue is that queries that seem rather simple and computable may never terminate. For example consider the query "list all documents reachable from o", and assume that o is the head of a long list of documents. The program evaluating the query starts from o and follows the links. However, it is possible that while the query program traverses the list, more and more documents are added to its end. If the reading is not faster than the appending, the computation will never terminate and the user will get the impression of dealing with an infinite list. This in spite of the fact that, at each point in time, the set of documents in the Web is finite.

Observe that not all the queries suffer from this non-terminating behavior. Consider the query "list all documents reachable from o by a path of distance less than 2". Suppose it is evaluated by a program that reads o, lists the URL's of all its outgoing links, reads the document for each such URL (if it exists), and repeats the process once more for the fetched documents. This program will always terminate, no matter how the Web changes while it runs. Note, however, that the answer may contain mixed data from various states of the Web – some of the documents o was pointing to may have been deleted after being read, some non-existing documents may have been added just before their URL was tested, etc. So the program does not necessarily return an answer that reflects the state of the Web as of some specific point in time.

This section studies the effect of the dynamic environment on queries. In particular, we focus on the effect on the computability and termination of queries. We assume a local-store based architecture where each document is fetched from the Web just once and cached locally if needed. In our Web machines this will be reflected by restricting the oracle to be called at most once on each oid. (So if several accesses are needed, the machine has to record the oracle answer on its working tape).

6.1. Databases and Queries

We next redefine our data model, and the notions of computable and eventually computable queries. Next we study the properties of queries in this dynamic context.

Definition 15 A *dynamic Web database schema*, is a modification of a Web database schema where instead of the single node and link relation schemes, there is an infinite sequence of relation schemes $N_1, L_1, N_2, L_2, \ldots$. An *instance* of a dynamic Web database is a mapping associating a finite set of tuples with every relation name in DB, finite sets of node objects with each of the N_i's, and finite sets of link objects with each of the L_i's, so that under the mapping, (DB, N_i, L_i) is a Web database instance, for all $i > 0$.

We assume that each of the N_i, L_i pairs reflects the state of the Web at some point in time, and that the sequence records the different states of the Web over time. We ignore here the issue of time granularity. Consecutive pairs may be mapped to the same instance to reflect the fact that the Web did not change in that time period, or may differ greatly to reflect the fact that many changes took place.

When a query is computed on the Web, its result may depend on how fast the query is evaluated – the faster it is evaluated, the fewer changes will be reflected in the result. The user often has no control or knowledge of the computation speed, which can depend on many parameters such as network or server load. Thus, at the abstract level, the user can view a query as a nondeterministic mapping. We want our model of computation to capture this nondeterminism and highlight the relationship between the computation speed and the query result. We incorporate these ideas into our definition of dynamic Web machines as follows.

Definition 16 A *dynamic Web machine* is a slightly modified Web machine:

1. The oracle input tape contains an encoding of the (infinite) sequence of relation pairs (N_1, L_1), (N_2, L_2), ...
2. The oracle is called at most once on each oid, and
3. At the beginning of the computation, the oracle uses (N_1, L_1). On each call it can decide nondeterministically whether to continue answering using the current (N_i, L_i) or switch to (N_{i+1}, L_{i+1}).

Each computation sequence reflects the relative speeds of computation and Web change. For example, the sequence where all calls are evaluated against (N_1, L_1) corresponds to a very fast computation that manages to complete all access to the Web before any changes take place (or alternatively a computation where concurrency control is employed). A sequence where all the calls are against distinct (N_i, L_i) pairs corresponds to the opposite case, where the rate of Web changes is as fast as the document fetching speed. Sequences where some of the calls are against the same (N_i, L_i), and some against distinct ones, reflect changing relative speeds.

Definition 17 A *(eventually computable) dynamic Web query,* is a non-deterministic mapping, i.e. a relation, from dynamic Web database instances to (possibly infinite) sets of tuples over the values in the instances, that is C-generic and (eventually) computable by a dynamic Web machine. "Computable by a dynamic Web machine" here means: for each input, all possible computations of the dynamic Web machine terminate, each possible output of the query is computed by some computation, and each computation computes some possible output. "Eventually computable by a dynamic Web machine" here means: for each input, and each (possibly infinite) output, there is a computation of the machine that eventually computes this output, and every computation eventually computes some output[†].

This definition differs from Definitions 3 (Web queries) and 8 (eventually computable Web queries) in several ways. First, the mapping is now non-deterministic, since we want to capture all possible outputs, which depend on the relatives speeds of query execution and update. Second, the mapping applies to dynamic Web instances, not static ones. Third, we allow each output to be an infinite set of tuples, because queries such as "list all documents reachable from o" may return an infinite set of nodes if updates are occurring faster than query processing.

Consider the query above, "list all documents reachable from o", and "list all documents reachable from o by a path of length less than 2," discussed at the beginning of this section. Intuitively, the second one can be computed by a dynamic Web machine such that all its possible computation sequences terminate. Thus it is a dynamic Web query. On the other hand, any machine for the first one will have some infinite computation sequences, so it is only an EC dynamic Web query. (All this will be made more precise, and formally proved, below).

All the computation sequences of the dynamic Web queries always terminate no matter how much the Web changes, or how fast the computation is. This is not the case for the EC dynamic Web queries . Nevertheless, some of the EC dynamic Web queries behave more nicely than others. Of particular interest are queries that guarantee computability when the computation is relatively fast compared to the update rate, as suggested in the following definition.

Definition 18 A *semi-computable* dynamic Web query, denoted SC dynamic Web query, is an EC dynamic Web query that can be computed by some dynamic Web machine such that every convergent computation terminates. A convergent computation of a dynamic Web machine is one where there is some finite k such that the oracle never switches to (N_i, L_i) with $i > k$.

Intuitively this means the user is guaranteed that, if the query is executed with a sufficiently fast engine, the computation terminates. Note that the engine can start working slowly (so the computation may be affected by changes in the Web), but if at some point it accelerates enough

[†] *eventually computes* is as in Definition 8.

so that all needed data from the Web are retrieved before any more Web changes happen, then the computation is guaranteed to halt. In concurrency control terms this means that such queries can start running without concurrency control – it suffices that concurrency control is employed at *some* point of the computation to guarantee termination.

To clarify things, here are a few examples.

- "List all nodes reachable from o" is an example of a SC dynamic Web query. If at some point the query engine accelerates enough so that all computation is done before any further updates occur, only nodes listed so far plus nodes reachable from them in the current state of the Web are output. Thus the computation terminates.

- On the other hand, "list all the nodes on the Web" is not an SC dynamic Web query, but only an EC dynamic Web query. This is because, even in the static case, when no changes at all are made to the Web (this is the simplest convergent computation), the machine does not halt.

- Finally, as in the static case, a query like "list all nodes with no incoming edges" is neither an SC nor an EC dynamic Web query. Again, this is because even when the Web does not change at all, the query cannot be eventually computed.

6.2. Dynamic vs. Static Queries

Query engines for the Web, such as WebSQL[13], W3QL [9], etc., provide query languages to specify user requests. The semantics of such engines is usually defined with respect to a static Web, i.e. what should the query result be if evaluated on some given Web database instance. However, in practice, when the program produced by the query compiler is executed, it runs in a dynamic environment where the Web keeps changing. Speaking in terms of Web machines, this means that the query is compiled into an ordinary (static) Web machine, but, when executed, the oracle in the machine is replaced by a dynamic one, and the dynamic machine is run instead.

What is the relationship between the original (static) query and the actual (dynamic) query being evaluated? This is the question studied below. First we need some auxiliary definitions.

Definition 19 A dynamic Web machine M *simulates* a Web machine M' if M is obtained from M' by replacing its oracle by a dynamic one.

Definition 20 A (EC) dynamic Web query Q *simulates* a (EC) Web query Q' if there is a dynamic Web machine M for Q, and a Web machine M' for Q', such that M simulates M'.

Note that this requires M' to call the oracle on each document only once. This is not a problem since every Web machine can be modified to satisfy this requirement by storing the answers of the oracle on the working tape and checking before each call if data on the given oid is already available.

We are now ready to state the relationship between queries, as defined by the user having a static view of the Web in mind, and the actual queries being executed in the dynamic environment. Not surprisingly, if the Web is not being updated, the actual execution computes exactly what the programmer intended.

Proposition 11 *If Q simulates Q', Q returns the same answer as Q' when given the same input and a static Web. That is, on input $(DB, (N_1, L_1), (N_2, L_2), ...)$ where $N_i = N_1, L_i = L_1$ for all i, Q computes a unique result that is the same as that of Q' on (DB, N_1, L_1).*

Proof. If Q simulates Q' then there is a dynamic Web machine M computing Q that is obtained from a regular Web machine M' for Q' by replacing the oracle by a dynamic one. But, if all the Webs in the dynamic input are identical, then the fact that the oracle switches between the Webs is insignificant: the oracle returns the same answer no matter which Web in the sequence it used, hence all the possible computations of the dynamic machine are identical up to the switches of the oracle, and the output produced by all of them is the same as the one produced by the computation where the oracle stays at the first Web. Thus, the same as in the static case. □

What happens when the Web is being updated? We focus below on computability issues and explain how they are affected by updates. The first theorem says that computable Web queries are precisely the class of queries whose computation, in a dynamic environment, is guaranteed to terminate provided that the query engine accelerates enough at some arbitrary point in the computation.

Theorem 12 *Every SC dynamic Web query simulates some Web query, and every Web query is simulated by some SC dynamic Web query.*

Proof. *Direction 1:* Given SC dynamic Web query Q, let M be a dynamic Web machine for Q where every convergent computation path terminates. Change the oracle to a static one. The resulting machine M' is a Web machine for some Web query, because its computation is the same as that of M in the run when $N_i = N_1, L_i = L_1$ for every i, therefore it terminates and computes a generic function of the input Web.
Direction 2: Given a Web query Q', we consider a specific Web machine M' for Q' that works in two phases: First it retrieves all the nodes reachable from values in base relations and the set of constants C, and then computes the query on this sub-Web. (So the first phase involves oracle calls, while the second part is an ordinary Turing machine computation on the nodes and links retrieved in the first phase). Note that such a machine M' exists for each Web query by Theorem 2.

Now, consider the dynamic Web machine M obtained by replacing the oracle of M' with a dynamic oracle. In each of the possible computations, M starts by attempting to compute the reachability set, i.e. retrieves all the nodes with ids in DB and C, then the nodes pointed by them, etc. Due to the dynamic environment, some of the computation sequences for this may not terminate. Note however that for all the convergent sequences this phase must terminate – this is because once the oracle stops switching Webs, we are back to simple computation of the reachability set, in the current Web, starting from the node ids retrieved so far. Since each of the individual Webs is finite, this is guaranteed to terminate. So the only sequences where the first phase may not terminate are the non convergent ones.

Now, if the first phase is completed, then the machines moves to the second phase which is computing the query w.r.t the nodes and links obtained in the first phase.

To conclude, the mapping computed by the machine is the following: For computation branches where the first phase terminates (and in particular for all the convergent branches) the output is the result of Q w.r.t the retrieved nodes and links. For all the other branches the result is empty (since the machine gets stuck in the first phase and never gets to the stage of producing an output). Clearly this mapping is generic and eventually computable by the given machine, and also since the infinite computations happen only in non-convergent branches, the query is SC. □

The next theorem deals with eventually computable Web queries. It says that EC Web queries are precisely the queries that can be run in a dynamic environment in a way that guarantees that all tuples in the result will eventually be output.

Theorem 13 *Every EC dynamic Web query simulates some EC Web query and every EC Web query is simulated by some EC dynamic Web query.*

Proof. *Direction 1:* Given an EC dynamic Web query Q, and any Web machine M for Q, change the oracle to a static one. The resulting machine M' is a machine for some EC query because its computation is the same as that of M on the branch where $N_i = N_1, L_i = L_1$, for every i. Therefore it eventually computes a generic function of the initial web.
Direction 2: The proof is similar to that of the previous Theorem. Given an EC Web query Q', we consider a specific Web machine M' that works as follows. First it computes the set of all nodes reachable using values from the base relations and C and evaluates Q on this sub-Web, and outputs the result. Next it starts enumerating oid's and, in parallel with the enumeration, computes the

reachability set of each oid, and every time it completes the computation of such a reachability set, takes the union of all the reachability sets computed so far (plus the initial one of DB and C), evaluates Q on this sub-Web, and outputs the new tuples in the result. Note that such a machine M' exists for all EC Web queries because of their monotonicity. (See Theorem 5.) Now, consider the dynamic Web machine M obtained by replacing the oracle of M' with a dynamic oracle. What M does in each of the possible runs is, first, to compute the reachability set of $DB+C$ for this specific sequence of oracle responses. Then, if this terminates, M effectively enumerates the result of evaluating the query on the subgraph containing all the nodes with finite reachability set (according to the oracle responses in the run). Clearly this mapping is generic and EC by the given machine.□

We considered above semi-computable and eventually-computable dynamic Web queries. What about the computable ones?

Theorem 14 *Every dynamic Web query simulates some Web query, but not conversely, i.e. there are Web queries that cannot be simulated by any dynamic Web query.*

This means that there are some Web queries such that no program for them is guaranteed to terminate when run in a dynamic environment. Furthermore, the following result shows that even if the query programmer is aware of the fact that the query will be run in a dynamic environment, writing an appropriate program that terminates on all inputs may be an impossible task. We do this by establishing the following.

Proposition 15 *There are Web queries Q' such that there is no dynamic Web query Q that is equivalent to Q' when given the same input and no changes to the Web. (i.e. on every input $(DB, N_1, L_1, N_2, L_2, ...)$ where $N_i = N_1, L_i = L_1$ for all i, Q computes a unique result that is the same as that of Q' on (DB, N_1, L_1).)*

We prove Theorem 14 and Proposition 15 simultaneously. The first direction for Theorem 14 is trivial (Simply take the machine of the dynamic query and replace the oracle by a static one). For the other direction, it suffices that we prove Proposition 15. (The second direction of Theorem 14 will then follow immediately from Proposition 11.) We gave above an intuitive explanation why a query like "all nodes reachable from o" cannot be evaluated in a dynamic environment. To prove Proposition 15, we now show this fact formally.

Proposition 16 *There is no dynamic Web query Q that, given a static Web as input, computes the reachability set of a node o in the input.*

Proof. We prove the Proposition by contradiction. Assume there exists such a query Q. We look at all the possible machines for this query, and for each one build an input on which at least one of the runs will not halt. This will contradict the fact the query is a dynamic Web query . Given a machine M, we look at a possible run and dynamically build the input on which the machine does not halt as follows: We start running the machine and see what is the first oid i on which it calls the oracle. We pick the first Node relation N_1 so that it contains some node o with id i, and the Link relation L_1 so that it contains some outgoing link from o pointing to some destination node with id $i' \neq i$. Now, we start running the machine and on each call to the oracle with a new id, we make the oracle switch to a new Web containing all the nodes in the previous Webs plus a new node having the id used in the call plus a link from the node to a destination with a new oid not seen so far.

Note that at each point of this run, the machine behaves in the same way as it would behave when running on any (non changing) Web containing all the nodes and links in the sub-Web seen so far. This Web contains at least one URL reachable from o on which the oracle was not called yet, (the one pointed by the last node on which the oracle was called). If the machine indeed computes the reachability query for the static Web, it cannot halt without calling the oracle at least once more to check if a node with the pointed id exists. So the run of the machine on the dynamic input will also attempt a least one more call. Since this is the case at each point in the run, the machine never halts for this specific input. A contradiction. □

This shows that some Web queries cannot be simulated by dynamic Web queries, hence they cannot be run in a dynamic environment safely, i.e., with a termination guarantee. Clearly, some queries can (such as "all nodes reachable from o by a path of length smaller then k" for some fixed k). Characterizing the class of Web queries that can run safely is an open problem. However we can present some sufficient conditions. For example, consider the Web calculus presented in the previous section. *Star-free* path expressions are those that use only the concatenation and union regular expression operators, that is, they do not use the Kleene closure.

Proposition 17 *All queries expressed by safe Web calculus formulas where all path expressions are star-free can be simulated by dynamic Web queries.*

Proof. Note that any occurrence of an atom of the form $Path(x, R, y)$ in a safe formula, where R is star-free, can be replaced by a subformula that does not use the $Path$ predicate, preserving safety. After this transformation, a simple induction on the structure of the resulting formula establishes the theorem. □

Thus it is possible to compile any such query into a program that will run safely in a dynamic environment, i.e. with no non-termination concerns. In fact, the programs generated by the implemented WebSQL compiler[13] for these queries are such terminating programs.

7. FORMS

So far we ignored for simplicity the issue of forms. We sketch below how they fit into the model and how our results extend to this more comprehensive model.

7.1. Data Model

Recall from Section 2 that forms are modeled as documents with parameterized outgoing links. These parameterized links are described in a relation L_p having the same structure as the links relation L, with an additional attribute *param_set*, i.e. $L_p[source, param\_set, destination, \ldots]$ The exact description of each parameter set is stored in the relation P using tuples of the form $[param\_set : t_1, param\_name : t_2, param\_val : t_3]$. So a (static) Web database has the form $W = (DB, N, L, L_p, P)$. We assume that the domain of set id's and of possible parameter names and values is infinite and disjoint from the domain of the other attributes.

We also assume that for each form, the set of possible parameter names and parameter values that potentially lead to actual documents is computable, given the form. This seems a reasonable assumption since in practice they can often be determined from the form structure and the size of attributes in it.

An *instance* of a Web database W is a mapping associating a finite set of tuples with each of the relations, such that the following holds: (1) *id* is a key in relation N, (2) there is a referential integrity constraint between the *source* attribute in L and L_P, and the *id* attribute in N, and (3) there is a referential integrity constraint between the *param_set* attribute in P to the one in L_p.

7.2. Queries

Queries (and eventually computable queries) were defined above using Web machines. In the context of forms, we slightly modify the oracle in the machine to support parameterized links:

1. The oracle input tape now also contains an encoding of the relations L_p and P.
2. When called, the oracle expects to find on its working tape a sequence of the form $oid\#plist$, where $plist$ is either empty or is a sequence of the form $pnam_1\#pval_1\#...\#pnam_n\#pval_n$.

3. If *plist* is empty, the oracle works as before and returns the (encoded) node tuple from N, and the tuples in L describing its outgoing links. Otherwise, the oracle returns the (encoded) node tuple from N, and the L_p tuples of its outgoing links such that the associated parameter set (according to P) is exactly the one described by *plist*.

(EC) Web queries are now defined as before, except that the Web instances are now over the extended schema, and the Web machine being used is the extended one described above. Also, note that the values of the attribute *param_set* are used only to encode sets of parameters, and are not really part of the data. We thus consider as (EC) Web queries only the mappings from Web instances to sets of tuples over the values in the instance, not including this attribute.

In the dynamic case, a Web database contains a sequence of relations

$$N_1, L_1, L_{p_1}, P_1, N_2, L_2, L_{p_2}, P_2, ...$$

and the oracle in the dynamic Web machine switches between them. Dynamic Web queries, EC dynamic Web queries and SC dynamic Web queries are defined as before using this machine.

It is now straight-forward to extend Theorems 2, 5, 6, 12, and 14 to handle forms.

8. CONCLUSION AND FURTHER WORK

We have given a new model of query and computation on the Web, closely related to, but significantly different from, the model of Abiteboul and Vianu [2]. We have separated the two reasons for non-terminating behavior of Web queries: navigational access to data and lack of concurrency control, and analyzed their impact.

Cost is an important aspect of query evaluation. One of the contributions of our computation model is in providing a good starting point for a theory of query complexity in the Web context. The conventional approach in database theory is to estimate query evaluation time as a function of the size of the database. Viewing the Web as infinite makes such an estimation problematic. Even if the Web is viewed as a finite database, simply adapting the standard complexity measures does not seem adequate – in the Web context, it is not realistic to try to evaluate queries whose complexity would be considered feasible in the usual theory, such as polynomial or even linear time.

For a query to be practical, it should not attempt to access too much of the network. Query analysis thus involves, in this context, two tasks: (1) Estimation of what part of the network may be accessed by the query. In terms of our computation model, this means estimating how many oracle calls are needed to evaluate a query, as a function of the size of the base relations, the size of the Web, and perhaps some bounds on Web parameters such as maximal size of documents, maximum outdegree, etc. (2) The cost of the query can then be analyzed in traditional ways as a function of the size of the base relations plus the size of the minimal sub-network that needs to be accessed.

Note that this separation lets us to take into consideration not only how much of the Web is accessed, but also the cost of accessing it. With the current Web architecture, the cost of document access is affected by document properties (e.g. size) and by the cost of communication between the site where the query is being evaluated and the site where the document is stored. This can be modeled by pricing oracle calls according to these parameters.

Preliminary work on complexity of Web-queries in the context of a static Web was presented in [13], where the notion of *query locality* was introduced to characterize the size of the network being accessed. Syntactic restrictions on WebSQL queries were presented to capture various locality classes. The case of EC queries and dynamic Web still needs to be studied.

Acknowledgements — We thank David Toman for his comments on an earlier version, and the anonymous referees for their detailed suggestions. This work was supported by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario. The work of the second author was supported by the Israel Academy of Sciences and Humanities. A preliminary version of this paper appeared in PODS'97 [14].

REFERENCES

[1] S. Abiteboul, C.H. Papadimitriou, and V.Vianu. The power of the reflective relational machine. In *Proceedings IEEE Symposium on Logic in Computer Science*, pp. 230–240, Paris, France (1994).

[2] S. Abiteboul and V. Vianu. Queries and computation on the web. In *Proceedings of the International Conference on Database Theory (ICDT)*, pp. 262–275, Delphi, Greece (1997).

[3] S. Abiteboul and V.Vianu. Generic computation and its complexity. In *Proceedings of the ACM STOC*, pp. 209–219 (1991).

[4] Paolo Atzeni, Giansalvatore Mecca, and Paolo Merialdo. To weave the web. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp.206–215, Athens Greece (1997).

[5] P. Brown and M. Stonebraker. BigSur: a system for the management of earth science data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 720–728, Zürich, Switzerland (1995).

[6] Ashok K. Chandra and David Harel. Computable queries for relational data bases. *JCSS*, **21**(2):156–178 (1980).

[7] Digital Equipment Corporation. Alta Vista: Main Page. `http://altavista.digital.com` (1998).

[8] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Record*, **27**(3):59–74 (1998).

[9] D. Konopnicki and O. Shmueli. W3QS: a query system for the world wide web. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 54–65, Zürich, Switzerland (1995).

[10] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. A declarative language for querying and restructuring the Web. In *Proceedings of 6th. International Workshop on Research Issues in Data Engineering, RIDE '96*, pp. 12-21, New Orleans, Louisiana (1996).

[11] S. Lawrence and C.L. Giles. Searching the world wide web. *Science*, **280**(4):98–100 (1998).

[12] Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing semistructured data with *FLORID*: A deductive object-oriented perspective. *Information Systems*, **23**(8):589–613 (1998).

[13] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. *International Journal on Digital Libraries*, **1**(1):54–67 (1997).

[14] Alberto O. Mendelzon and Tova Milo. Formal models of web queries. In *Proceedings of 16th Symp. on Principles of Database Systems - PODS 97*, pp. 134–143 (1997).

[15] Chee K. Yap Richard Hull. The format model: A theory of database organization. *J. ACM*, **31**(3):518–537 (1984).