A Query Language and Processor for a Web-Site Management System

Mary Fernandez Daniela Florescu Alon Levy Dan Suciu

April 18, 1997

1 Introduction

We have designed a system, called STRUDEL (to be demonstrated at SIGMOD-97 [FFK+97]), which applies familiar concepts from database management systems, to the process of building web sites. The main motivation for developing STRUDEL is the observation that with current technology, creating and managing large sites is tedious, because a site designer must simultaneously perform (at least) three tasks: (1) choosing what information will be available at the site, (2) organizing that information in individual pages or in graphs of linked pages, and (3) specifying the visual presentation of pages in HTML. Furthermore, since there is no separation between the physical organization of the information underlying a web site and the logical view we have on it, *changing* or *restructuring* a site are unwieldy tasks.

In STRUDEL, the web site manager can separate the logical view of information available at a web site, the structure of that information in linked pages, and the graphical presentation of pages in HTML. First, the site builder defines independently the data that will be available at the site. This process may require creating an *integrated* view of data from multiple (external) sources. Second, the site builder defines the *structure* of the web-site. The structure is defined as a *view* over the underlying information, and different versions of the site can be defined by specifying multiple views. Finally, the graphical representation of the pages in the web site is specified.

This abstract describes the query language and query processor that lie at the heart of the STRUDEL system. In STRUDEL, we model the data at the different levels as graphs. That is, the data in the external sources, the data in the integrated view and the web-site itself are modeled as graphs. A graph model is appropriate because site data may be derived from multiple sources, such as existing database systems and HTML files. Consequently, our system requires a query language for (1) defining the integrated view of the data, and (2) defining the structure of web sites. An important requirement of our query language is that it be able to *construct* graphs. Our query processor needs to be able to answer queries that involve accessing different data sources. Even though we model the sources as containing graphs, we cannot assume they have a *uniform* representation of graphs. Hence, our query processor needs to *adhere* to possible limitations on access to data in the graphs, and should be able to *exploit* additional querying capabilities that an external source may have.

We have designed a general framework for processing STRUDEL queries over multiple unstructured data sources, and are designing optimizations that use the capabilities of external sources whenever possible. We first describe STRUDEL's data model and query language. Our examples show how simple site-management tasks are implemented in STRUDEL's query language. We then describe the query processor, its multiple-level representation of queries, and how these representations are translated into source-specific execution plans.

2 Data Model and Query Language

In every level of the STRUDEL system, data is viewed uniformly as a graph. At the bottom-most level, data is stored in STRUDEL's *data graph* repository or in external sources. External sources may have a variety of formats, but each is translated into the graph data model by a wrapper (see Figure 1). STRUDEL's graph model is similar to that of OEM [PGMW95]. A data graph contains *objects* connected by directed edges labeled with string-valued *attribute names*. Objects are either nodes, carrying a unique object identifier (oid), or are atomic values, such as integers, strings, files, etc. STRUDEL also provides named *collections* of objects, i.e., sets of oids: technically speaking these are redundant, since every collection can be represented as a wide subtree, but we included them in the data model for convenience.

Strudel's central data repository is the *data graph*. It describes the logical structure of all the information available at that site, and is obtained by integrating information from the various external sources. From the data graph, a site builder can define one or more *site graphs*; each represents the logical structure of the information displayed at that site (i.e., a node for every web page and attributes describing the information in the page and links between pages). There can be several site graphs, because one site may offer several views of its data; each site graph corresponds to one view. Finally, the HTML generator constructs a browsable HTML graph from a graph site. The HTML generator creates a web page for every node in the site graph, using the values of the attributes of the node. In this paper we focus on generating the site graphs and do not discuss the HTML generator.

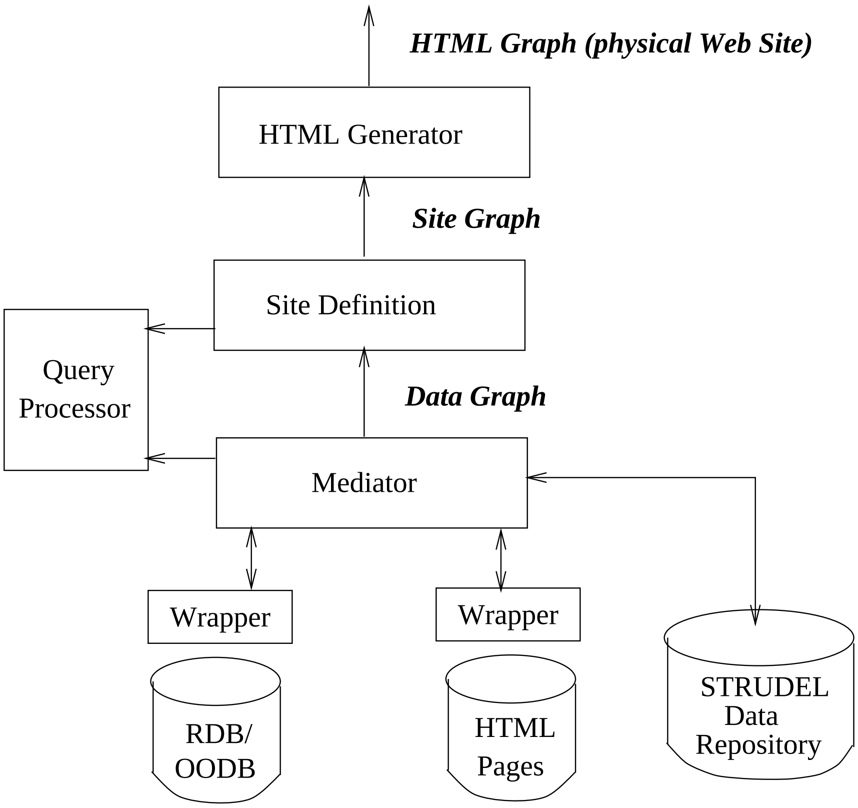


Figure 1: STRUDEL Architecture

In STRUDEL, we need to query and/or to transform graphs: (1) at the *integration* level, when data from different external sources is integrated into the data graph, and (2) at the *site-graph definition* level, when site graphs are constructed from a data graph. In addition, STRUDEL's design enables us to provide an interface for ad-hoc queries over a web site. We use a common query and transformation language, STRUQL(Site TRansformation Und Query Language), at all three levels. We describe STRUQL's core fragment next.

Syntax. A STRUQL query has the form:

where $C_1, \ldots, C_k$,
[create $N_1, \ldots, N_n$]
[link $L_1, \ldots, L_p$]
collect $G_1, \ldots, G_q$

For example, the following query returns all PostScript papers directly accessible from home pages:

where $HomePages(p), p$ "$Paper$" $q, isPostScript(q)$
collect $PostscriptPages(q)$

Here $HomePages$ is a collection, "$Paper$" is an edge label, and $isPostScript$ is a predicate testing whether node q is a PostScript file. The condition p "$Paper$" q means that there exists an edge labeled "$Paper$" from p to q. The query constructs a new collection, $PostscriptPages$, consisting of all answers.

In the general syntax, each condition $C_1, \ldots, C_k$ in a where clause either (1) tests collection membership, e.g., $HomePages(p)$, or (2) is a regular path expression, e.g., p "$Paper$" q, or (3) is a built-in or external predicate applied to nodes or edges, e.g., $isPostScript(q)$, $x > y$, or (4) is a boolean combination of the above. A condition of the form "$x\ R\ y$" means that there exists a path from node x to node y that matches the regular path expression R. Regular path expressions are more general than regular expressions, because they permit predicates on edges and nodes. For example "$isName*$" is a regular path expression denoting any sequence of labels such that each satisfies the $isName$ predicate. In particular, $true$ denotes any edge label, and $true*$ any path; we abbreviate the latter with $*$. Other operators include path concatenation and alternation; the grammar for regular path expressions is: $R ::= Pred \mid Pred; Pred \mid R.R \mid (R|R) \mid R*$. The expression "$Pred1; Pred2$" matches any edge, node pair whose edge satisfies $Pred1$ and whose node satisfies $Pred2$. $Pred$ is shorthand for the idiom "$Pred; true$", which ignores the node value. For example,

where $Root(p)$, $p\ (true; not(HomePage))*\ q$
collect $Result(q)$

returns all nodes q accessible from paths that begin at the root and that do not go through any $HomePage$. We assume in this paper that $Root$ is a collection consisting of the graph's single root node.

StruQL is novel, because it extends these simple regular-expression based selections with the creation of new graphs from existing graphs; its create, and link clauses specify new graphs. The following example copies the input graph and adds a "$Home$" edge from each node back to the root:

where $Root(p),\ p * q,\ q\ l\ q'$
create $N(p),\ N(q),\ N(q')$
link $N(q)\ l\ N(q'), N(q)$ "$Home$" $N(p)$
collect $NewRoot(N(p))$

N is a Skolem function creating new object oids. The query first finds all nodes q reachable from the root p (including p itself) and all nodes q' directly accessible from q by one link labeled l. For each node q and q', it constructs new nodes $N(q)$ and $N(q')$: in effect, this copies all nodes accessible from the root once. The query adds a link l between any pair of nodes that were linked in the original graph and adds a new $Home$ link that points to the new root. Finally, it creates an output collection $NewRoot$ that contains the new graph's root.

A similar query produces a site graph, i.e., a view of the input graph, called $TextOnly$, that excludes any nodes that contain image files:[1]

where $Root(p),\ p * q,\ q\ l\ q', not(isImageFile(q'))$
create $N(p),\ N(q),\ N(q')$
link $N(q)\ l\ N(q'),$
collect $TextOnlyRoot(N(p))$

StruQL has other constructs for supporting regular path expressions that are omitted from this abstract.

[1]This example is inspired by an inconsistency in the CNN web site `http://www.cnn.com`. The site provides a link to a text-only version, but only for the root page. Surprisingly, the following links point again to pages with images.

Semantics. A distinguishing feature of STRUQL is that its semantics can be described in two stages. The *query stage* depends only on the where clause and produces all possible bindings of variables to values that satisfy all conditions in the clause; its result is a relation with one attribute for each variable. The *construction* stage constructs a new graph from this relation, based on the create, link, collect clauses. We explain the details next.

We adopt an *active-domain* semantics for STRUQL. For a data graph G, let O be the set of all oids and L the set of all labels in G. Let V be the set of all node and edge variables in a query. The meaning of the where-clause is the set of assignments $\theta : V \to O \cup L$ that satisfy all conditions in the where clause. Each assignment maps node variables to O and edge variables to L. The meaning of the create $-$ link $-$ collect clauses is as follows. For each row in the relation, first construct all new node oids, as specified in the create clause. Assuming the latter is create $N_1, \ldots, N_n$, each N_i is a Skolem function applied to node oids and/or label values. By convention, when a Skolem function is applied to the same inputs, it returns the same node oid. Next, construct the new edges, as described in the link clause. We impose two semantic conditions on STRUQL's queries: first each node mentioned in link or collect is either mentioned in create or is a node in the data graph. Second, edges can only be added from new nodes to new or existing nodes; existing nodes are immutable and cannot have edges added to them. Thus, link $f(x)"A"f(y)$ and link $f(x)"A"y$ are legal, but link $x"A"f(y)$ is not, because the latter would alter the old graph. Finally, the semantic of the collect clause is obvious.

Under the active-domain semantics, *every* STRUQL query has a well-defined meaning. For example, the query below returns the complement of a graph, i.e., the result graph contains an edge labeled l between two nodes if and only if no such label exists between the corresponding nodes in the input graph:

```
where  not(p l q)
create f(p), f(q)
link   f(p) l f(q)
```

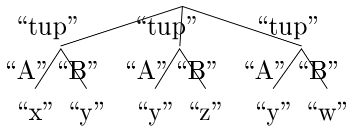
However, the active-domain semantics is unsatisfactory because it depends on how we define the active domain; the semantics changes if, for example, we compute the active domain only for the *accessible* part of the graph. The situation is similar to the domain independence issue in the relational calculus: there it is solved by considering *range-restricted* queries, which are guaranteed to be domain independent, i.e., their semantics does not change if we artificially change the active domain. We are currently specifying range-restriction rules for STRUQL.

Expressive power. STRUQL's regular path expressions, like those in LOREL and UnQL, require graph traversal and, therefore, the computation of transitive closure. The ability to compute the transitive closure of an input graph does not imply the ability to compute the transitive closure of an *arbitrary* binary or $2n$-ary relation. This is proven formally for UnQL [BDHS96]. Surprisingly, STRUQL *can* express transitive closure of an arbitrary relation as the composition of two queries[2]. For example, consider the tree-encoding of a binary relation $R(A, B)$ with attributes A and B, as shown below. We can compute all nodes reachable from "x" with two STRUQL queries. The first constructs the graph corresponding to the relation $R(A, B)$, and the second uses the regular expression $*$ to find all nodes accessible from the root. input is a keyword that specifies the input to a query:

[2]It follows from the result in [BDHS96] that a single where $-$link query cannot express transitive closure.

input (where $Root(r), r \text{ "}tup\text{" } s1, r \text{ "}tup\text{" } s2,$
$s1 \text{ "}A\text{" } x1, s1 \text{ "}B\text{" } y1$
$s2 \text{ "}A\text{" } x2, s2 \text{ "}B\text{" } y2$
$y1 = x2$
create $N(y1), N(x2)$
link $N(y1) \text{ "}bogus\text{" } N(x2)$
collect $NewRoot(N(\text{"}x\text{"})))$
where $NewRoot(x), x * N(y)$
collect $Result(y)$

A	B
“x”	“y”
“y”	“z”
“y”	“w”



We can prove that StruQL has precisely the same expressive power as first order logic extended with transitive closure [Imm87], FO+TC. That is, considering a two-sorted (with sorts Oid and $Label$), first order vocabulary $\tau = \{E, C_1, \ldots, C_k\}$, where $E(Oid, Label, Oid)$ is the edge relation and $C_1(Oid), \ldots, C_k(Oid)$ are the collections, then a boolean query over this vocabulary is expressible in pure $StruQL$ (i.e. without external predicates) if and only if it is expressible in FO + TC.

Related languages. The query stage of StruQL is similar to LOREL [QRS+95], but unlike LOREL, StruQL can construct a new output graph. This feature is strictly necessary in our application. Compared to UnQL [BDHS96], which can also construct new graphs, the construction part of StruQL is more powerful, as explained above. G [CMW87, CMW88, Woo88] is also a query language for graphs with regular path expressions, graph construction, and active-domain semantics: the graph construction part of G corresponds to the restriction of StruQL to unary Skolem functions.

3 Query Evaluation and Optimization

A naive evaluation strategy based on our active-domain semantics is prohibitively inefficient, because the semantics constructs an intermediate relation whose size may be much larger than that of the input graph. In general, such a strategy is also unnecessary, because transformation queries typically require at most one traversal of the input graph and may not require the computation of an intermediate relation at all. Assuming that all data is materialized in data graphs, we could apply query evaluation strategies for other semi-structured languages [QRS+95, BDHS96, MMM96] to StruQL and use optimization techniques that minimize or eliminate graph traversal [FS96]. Strudel data, however, may be stored in external sources, and therefore, we want evaluation strategies that adhere to limitations of external sources and can use the query capabilities of the sources whenever possible. Identifying such evaluation strategies is non-trivial, because although we model data in external sources as data graphs, the external sources may not support queries that traverse arbitrary paths through the data. In particular, we want evaluation strategies that can:

Push queries to external sources. Strudel data graphs can be created by integrating information from external sources. A simple strategy is to materialize external sources as data graphs and evaluate queries on the materialized graphs. A better strategy is to delay fetching of external data by pushing queries to the external sources. This is the approach taken by TSIMMIS using MSL [PGMU95, PAGM96], an integration language related to Datalog. For StruQL the problem is harder due to the presence of regular path expressions.

Exploit indexes. Indexes may be available for data graphs materialized in the data repository and for the external sources. We want our query processor to use these indexes whenever possible. The effective use of indexes when evaluating regular path expressions is an open problem.

Handle limited capabilities. The ability to traverse edges forward and/or backward depends on the capability of a data source. In some external sources, links may be traversed only in one direction. For example, the AT&T employee database requires certain attribute values such as last name or phone number, in order to extract any information. In our data model, this translates into certain edges being traversable in one direction (note that this is a generalization of the problem of limitation on binding patterns considered in [RSU95, LRO96]).

We believe that to address all these requirements, it is advantageous to apply different optimization strategies to different representations of a query, with the hope that at each level we can express different kinds of optimizations. We use three representations: (1) StruQL's abstract syntax, (2) automata that compute regular path expressions, and (3) a physical query plan that incorporates the capabilities of the data source. We show some examples of optimizations at each level, which essentially rewrite a query into an equivalent representation at the same level and/or translate the query into the representation at the next level. It is possible to combine such isolated optimization strategies using the techniques in [Flo96].

Abstract syntax level. At this level, we apply source-to-source transformations to reduce the number of edges that must be traversed in the input graph. We do this by finding common prefixes in multiple regular path expressions. For example, consider:

where $Root(p)$,
    p ("A"."B")$*$."A" q,
    p "A"."C"."D" r

We can rewrite the first regular expression into an equivalent expression:

where $Root(p)$,
    p "A".("B"."A")$*$ q,
    p "A"."C"."D" r

Next, we introduce two node variables, x and y:

where $Root(p)$,
    p "A" x,
    x ("B"."A")$*$ q,
    p "A" y,
    y "C"."D" r

Finally, if q and r are "independent", meaning that none of the clauses in link creates and edge between them, then we can collapse the x and y node variables into a single variable z, and get:

where $Root(p)$,
    p "A" z,
    z ("B"."A")$*$ q,
    z "C"."D" r

The resulting query is equivalent to the original, but it traverses each edge "A" accessible from the root once.

Automaton level. At this level, each regular expression is translated into a generalized NDFA. There is one automaton for each condition in the where clause. The simplest way to execute an automaton is to traverse it from its input state to its output state(s), but we consider also more complex traversals, which start at some arbitrary state(s), resulting in alternative physical execution plans. We will illustrate this in Figure 2, for the query:

where $C1(x), x(AB)+\ y,$
    $yCz, C2(y), z > 4$

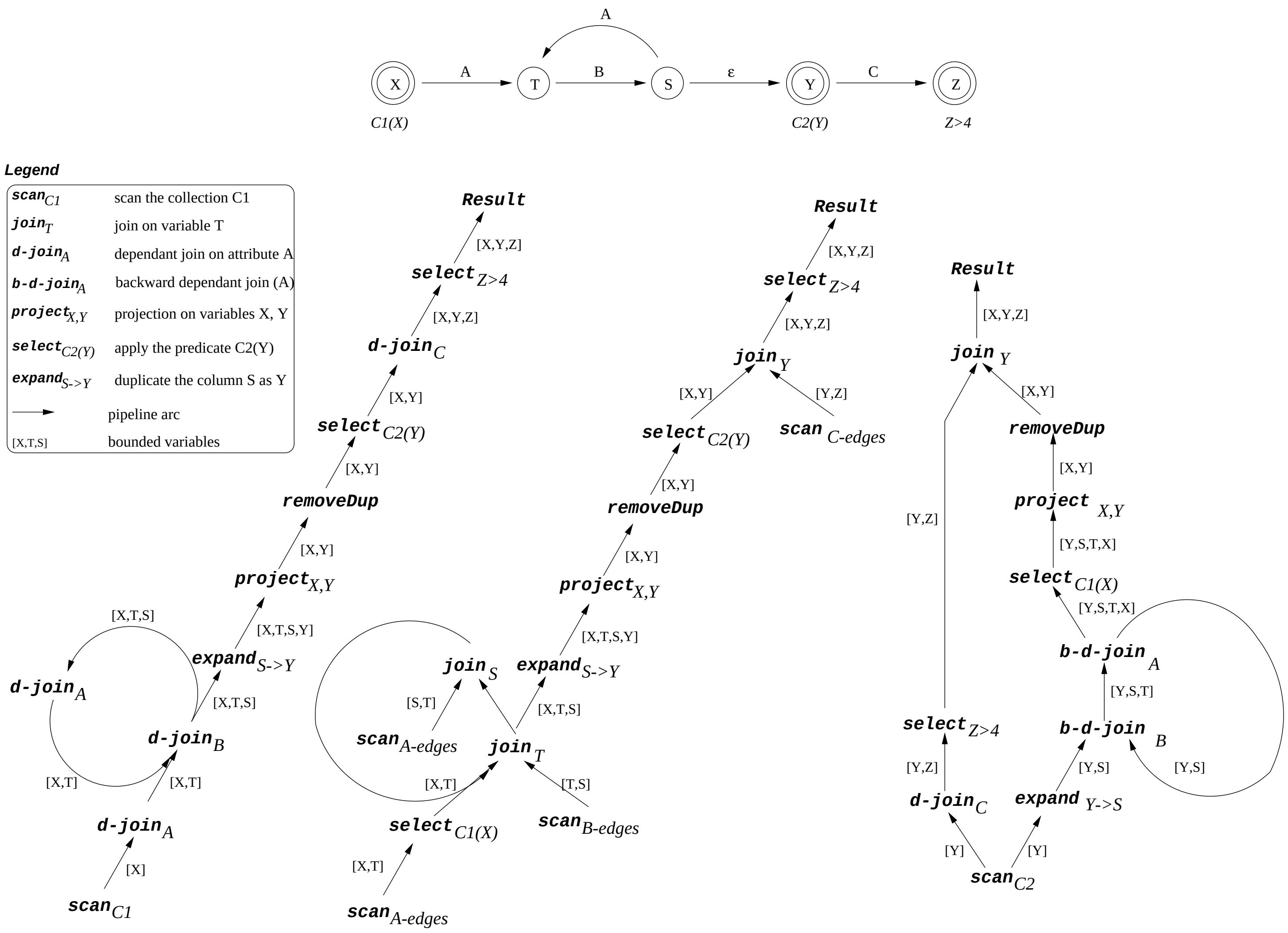


Figure 2: An NDFA and three physical plans.

Physical execution plan. The operators at the physical level are the same as those found in relational query plans, see Figure 2. The novelty is that the physical plans may have cycles. The first two plans in Figure 2 correspond to the left-to-right traversal of the automaton: they differ only in the fact that the first has a direct join while the second has a join. The third execution plan is more interesting. It corresponds to starting the automata in the state Y: this could be advantageous if, for example, we have fast access to the collection $C2$. Then the plan branches, searching for a C edge to Z on one hand, and for a backwards path to X in $C1$ on the other hand. At the end it has to join the two results based on Y.

Cluet and Moerkotte describe an algebra for querying semi-structured data [CM97]. They consider regular expressions as atomic operators in their algebra (as part of the *tree patterns* in the *scan* operator). This enables them to optimize queries at the select-project-join level. By contrast, we want to optimize inside regular expressions too, so our execution plans are more fine grained, and generally have cycles.

References

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, 1996.

[CM97] Sophie Cluet and Guido Moerkotte. Query processing in the schemaless and semistructured context. Manuscript, 1997.

[CMW87] I. Cruz, A.O. Mendelzon, and P.T Wood. A graphical query language supporting recursion. In *Proceedings of ACM SIGMOD Conf.*, San Francisco, California, May 1987.

[CMW88] I. Cruz, A.O. Mendelzon, and P.T Wood. G+: recursive queries without recursion. In *Proc. Second Int'l Conf. on Expert Database System*, Tysons Corner, Virginia, April 1988.

[FFK+97] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. STRUDEL - a web-site management system. In *SIGMOD*, Tucson, Arizona, May 1997.

[Flo96] Daniela D. Florescu. *Design and Implementation of the Flora Object Oriented Query Optimizer.* PhD thesis, Institut National de Recherche en Informatique et Automatique (INRIA), 1996.

[FS96] Mary Fernandez and Dan Suciu. Query optimizations for semi-structured data using graph schema (full version), 1996. Manuscript available from `http://www.research.att.com/info/{mff,suciu}`.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd VLDB Conference, Bombay, India.*, 1996.

[MMM96] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. In *Proceedings of the Fourth Conference on Parallel and Distributed Information Systems*, Miami, Florida, December 1996.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of VLDB*, September 1996.

[PGMU95] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *IEEE International Conference on Data Engineering*, pages 132–141, March 1995.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, March 1995.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructure heterogeneous information. In *International Conference on Deductive and Object Oriented Databases*, 1995.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Jose, CA*, 1995.

[Woo88] Peter T. Wood. *Queries on Graphs.* PhD thesis, University of Toronto, Toronto, Canada, M5S 1A1, December 1988. Available as University of Toronto Technical Report CSRI-223.