Efficient Queries over Web Views

Giansalvatore Mecca,[1] Alberto O. Mendelzon,[2] Paolo Merialdo[3]

[1] DIFA, Università della Basilicata, Potenza, Italy
[2] Dept. of Computer Science, University of Toronto, Canada
[3] Dip. Informatica e Automazione, Università di Roma Tre, Roma, Italy

Abstract. Large web sites are becoming repositories of structured information that can benefit from being viewed and queried as relational databases. However, querying these views efficiently requires new techniques. Data usually resides at a remote site and is organized as a set of related HTML documents, with network access being a primary cost factor in query evaluation. This cost can be reduced by exploiting the redundancy often found in site design. We use a simple data model, a subset of the Araneus data model, to describe the structure of a web site. We augment the model with link and inclusion constraints that capture the redundancies in the site. We map relational views of a site to a navigational algebra and show how to use the constraints to rewrite algebraic expressions, reducing the number of network accesses.

1 Introduction

As the Web becomes a preferred medium for disseminating information of all kinds, the sets of pages at many Web sites have come to exhibit regular and complex structure not unlike the structures that are described by schemes in database systems. For example, Atzeni *et al.* [4] show how to describe the structure of the well-known Database and Logic Programming Bibliography at the University of Trier [9] using their own data model, the ARANEUS data model.

As these sites become large, manual navigation of these hypertext structures ("browsing") becomes clearly inadequate to retrieve information effectively. Typically, ad-hoc search interfaces are provided, usually built around full-text indexing of all the pages at the site. However, full-text queries are good for retrieving documents relevant to a set of terms, but not for answering precise questions, e.g. "find all authors who had papers in the last three VLDB conferences." If we can impose on such a site a database abstraction, say a relational schema, we can then use powerful database query languages such as SQL to pose queries, and leave it to the system to translate these declarative queries into navigation of the underlying hypertext.

In this paper we explore the issues involved in such a translation. In general, a declarative query will admit different translations, corresponding to different navigation paths to get to the data; for example, the query above could be answered by:

1. Starting from the home page, follow the link to the list of conferences, from here to the VLDB page, then to each of the last three VLDB conferences, extract a list of authors for each, and intersect the three lists.
2. As above, but go directly from the home page to the list of database conferences, a smaller page than the one that lists all conferences.

3. As above, but go directly from the home page to the VLDB page (there is a link).
4. Go through the list of authors, for each author to the list of their publications, and keep those who have papers in the last three VLDB's.

If we use number of pages accessed as a rough measure of query execution cost, we see there are large differences among these possible access paths, in particular between the last one and the other three. There are over 16,000 authors represented in this bibliography, so the last access path would retrieve several orders of magnitude more pages than the others. Given these large performance differentials, a query optimizer is needed to translate a declarative query to an efficient navigation plan, just as a relational optimizer maps an SQL query to an efficient access plan. In fact, there is an even closer similarity to the problem of mapping declarative queries to network and object-oriented data models, as we discuss in Section 2.

To summarize, our approach is to build relational abstractions of large and fairly well-structured web sites, and to use an optimizer to translate declarative queries on these relational abstractions to efficient navigation plans. We use a simple subset of the ARANEUS data model (ADM) to describe web sites, augmenting it with *link constraints* that capture the redundancy present in many web sites. For example, if we want to know who were the editors of VLDB '96, we can find this information in the page that lists all the VLDB conferences; we do not need to follow the link from this page to the specific page for VLDB '96, where the information is repeated. We also use *inclusion constraints*, that state that all the pages that can be accessed using a certain path can also be accessed using another path. We use a *navigational algebra* as the target language that describes navigation plans, and we show how to use rewrite rules in the spirit of relational optimizers, and taking link and inclusion constraints into account, to reduce the number of page accesses needed to answer a query.

When a query on the relational views is issued, it is repeatedly rewritten using the rules. This process generates a number of navigation plans to compute the query; the cost of these plans is then estimated based on a simple cost model that takes network accesses as the primary cost parameter. In this way, an efficient execution plan is selected for processing the query.

Query optimization is hardly a new topic; however, doing optimization on the Web is fundamentally different from optimizing relational or OO databases. In fact, the Web exhibits two peculiarities: the *cost model* and the *lack of control over Web sites*: (*i*) *the cost model*: since data reside at a remote site, our cost model is based on the number of network accesses, instead of I/O and CPU cost, and we allocate no cost to local processing such as joins; (*ii*) *the lack of control over the site*: unlike ordinary databases, sites are autonomous and beyond the control of the query system; first, it is not possible to influence the *organization of data* in the site; second, the site manager inserts, deletes and modifies pages without notifying remote users of the updates.

These points have fundamental implications on query processing. The main one is that we cannot rely on auxiliary access structures besides the ones already built right into the HTML pages. Access structures – like *indices* or *class extents* – are heavily used in optimizing queries over relational and object-oriented databases [14]. Most of the techniques proposed for query optimization rely on the availability of suitable access structures. One might think of extending such techniques to speed up the evaluation of queries by, for example, storing URLs in some local data structures, and then using them in query evaluation. However, this solution is in general unfeasible because, after the data structures have been constructed, they have to be maintained; and since our system is not notified of updates to pages, the only way of maintaining these structures is to actually navigate the site at query time checking for updates, which in general has a cost comparable to the cost of computing the query itself.

We therefore start our analysis under the assumption that the only access structures to pages are the ones built right into the hypertext. Due to space limitations, we concentrate on the issue of mapping queries on *virtual* relational views to navigation of the underlying hypertext, and develop an algorithm for selecting efficient execution plans, based on a suitable cost function. In [10], we study the problem of querying *materialized* views, and show how the same techniques developed for virtual views can be extended to the management of materialized views.

Outline of the Paper The outline of the paper is as follows. We discuss related work in Section 2. Sections 3 and 4 present our data model and our navigational algebra; the problem of querying virtual views is introduced in Section 5; the rewrite rules and the optimization algorithm are presented in Section 6; Section 7 discusses several interesting examples. Due to space limitations, the presentation is mainly informal. Details can be found in [10].

2 Related Work

Query optimization Our approach to query optimization based on algebraic rewriting rules is inspired on relational and object-oriented query optimization (e. g., [18], [5]). This is not surprising, since it has been noted in the context of object-oriented databases that relational query optimization can be well extended to complex structures ([14], [7]). However, the differences between the problem we treat here and conventional query optimization, which we listed in the Introduction, lead to rather different solutions.

Optimizing path expressions Evaluating queries on the Web has some points of contact with the problem of optimizing *path-expressions* [22] in object-oriented databases (see, for example, [6], [14]). Since path-expressions represent a powerful means to express navigation in object databases, a large body of research about query processing has been devoted to their optimization. In this research, the focus is on transforming *pointer chasing* operations – which are considered rather expensive – into *joins of pointer sets* stored in auxiliary access structures, such as class extents [7], access support relations [8] and join indices [19], [20].

Although it may seem that a similar approach may be extended to the Web, we show that the more involved nature of access paths in Web sites and the absence of *ad-hoc* auxiliary structures introduces a number of subtleties. We compare two main approaches to query optimization: (i) the first one, that we might call a *"pointer join"* approach, is inspired on object-oriented query optimization: it aims at reducing link traversal by manipulating (joining) pointer sets; (ii) the second is what we call a *"pointer chase"* approach, in which links between data are used to restrict network access to relevant items. An interesting result is that, in our cost model, sometimes navigation is less expensive than joins. This is different from object-oriented databases, where the choice between the two is generally in favor of the former [6].

Relational Views over Network Databases The idea of managing relational views over hypertextual sources is similar to some proposals (e.g., [21], [13], [16]) for accessing network databases through relational views: links between pages may recall set types that correlate records in the network model. However, in these works the focus is more on developing tools and methods for automatically deriving a relational view over a network database, than on query optimization. More specifically, one of the critical aspects of accessing data in the Web – i.e., selecting one among multiple paths to reach data – is not addressed.

Indices in Relational Databases It has already been noted in the previous section that our approach extends to the Web a number of query optimization techniques developed in the context of relational databases. Another related issue is the problem of selecting one among several indices available for a relation in a relational database (see, for example, [15] and [12]); this has some points in common with the problem of selecting one among different access paths for pages in a Web site; however, paths in the Web are usually more complex than simple indices, and our cost model is radically different from the ones adopted for relational databases.

Path Constraints The presence of *path constraints* on Web sites is the core of the approach developed in [1]. The authors recognize that important structural information about portions of the Web can be expressed by constraints; they consider the processing of queries in such a scenario, and discuss how to take advantage of constraints. The fundamental difference with our approach is that we work with an intensional description of Web data, based on a database-like data model, while the authors of [1] reason directly on the extension of data.

3 The Data Model

Our data model is essentially a subset of ADM [4], the ARANEUS data model; the notion of *page-scheme* is used to describe the (possibly nested) structure of a set of homogeneous Web pages; since we are interested in query optimization, in this paper we enrich the model with *constraints* that allow reasoning about redundancies in a site, e.g., multiple paths to reach the same data. From this perspective, a *scheme* gives a description of a portion of the Web in terms of page-schemes and constraints. It is important to note that this description of the Web portion is usually *a posteriori*, that is, both the page-schemes and the constraints are obtained, not from a forward engineering phase, but rather from a *reverse engineering* phase, which aims at describing the structure of an existing site. This analysis is conducted by a human designer, with the help of a number of tools which semi-automatically analyze the Web in order to find regular patterns.

3.1 Page-schemes

Each Web page is viewed as an object with a set of attributes. Structurally similar pages are grouped together into sets, described by *page-schemes*. Attributes may have simple or complex type. Simple type attributes are *mono-valued* and correspond essentially to text, images, or *links* to other pages. Complex type, *multi-valued* attributes are used to model collections of objects inside pages, and correspond to lists of tuples, possibly nested.

The set of pages described by a given page-scheme is an *instance* of the page-scheme. It is convenient to think of a page-scheme as a nested relation scheme, a page as a nested tuple on a certain page-scheme, and a set of similar pages as an instance of the page-scheme. There is one aspect of this framework with no counterpart in traditional data models. There are pages that have a special role: they act as "entry-points" to the hypertext. Typically, at least the home page of each site falls into this category. In ADM *entry points* are modeled as page-schemes whose instance contains only one tuple.

To formalize these ideas, we need two interrelated definitions for types and page-schemes, as follows. Given a set of *base types* containing the types *text* and *image*, a set of *attribute names* (or simply *attributes*), and a set of *page-scheme names*, the set of WEB *types* is defined as follows (each type is either mono-valued or multi-valued):

- each base type is a *mono-valued* WEB *type*;
- LINK TO P is a *mono-valued* WEB *type*, for each page-scheme name P;
- LIST OF$(A_1 : T_1, A_2 : T_2, \ldots, A_n : T_n)$ is a *multi-valued* WEB *type*, if A_1, A_2, $\ldots, A_n$ are attributes and $T_1, T_2, \ldots, T_n$ are WEB types;

A *page-scheme* has the form $P(URL, A_1 : T_1, A_2 : T_2, \ldots, A_n : T_n)$, where P is a page scheme name, each A_i is an attribute, each T_i is a WEB type, and URL is the Universal Resource Locator of P, and forms a key for P.

An *entry-point* is a pair $(P,\ URL)$, where P is a page-scheme and URL is the URL for a page p which is the only tuple in the instance of P. As we suggested above, an *instance* of a page-scheme is a *page-relation*, i.e., a set of nested tuples, one for each of the corresponding pages, each with a URL and a value of the appropriate type for each page-scheme attribute. Entry points are page-relations containing a single nested tuple.

Note that we do not assume the availability of *page-scheme extents*: the only pages whose URL is known to the system are instances of entry points; any other page-relation can only be accessed by navigating the site starting from some entry point. It is also worth noting that, in order to see pages, i.e., HTML files, as instances of page-schemes, i.e., nested tuples, we assume that suitable *wrappers* [3, 2] are applied to pages in order to access attribute values.

We have experimented with our approach on several real-life Web sites. However, in this paper we choose to refer to a fictional site – a hypothetical university Web site – constructed in such a way as to allow us to discuss with a single and familiar example all relevant aspects of our work. Figure 1 shows some examples of page-schemes from such site.

3.2 Constraints

The hypertextual nature of the Web is usually associated with a high degree of redundancy. Redundancy appears in two ways. First, many pieces of information are replicated over several pages. Consider the Department example site: the name of a Department –say, *Computer Science*– can be found not only in the *Computer Science* Department page but also in many other pages: for instance, it is presumably used as an anchor in every page in which a link towards the department page occurs. Second, pages can be usually reached following different navigational paths in the site. To capture these redundancies so they can be

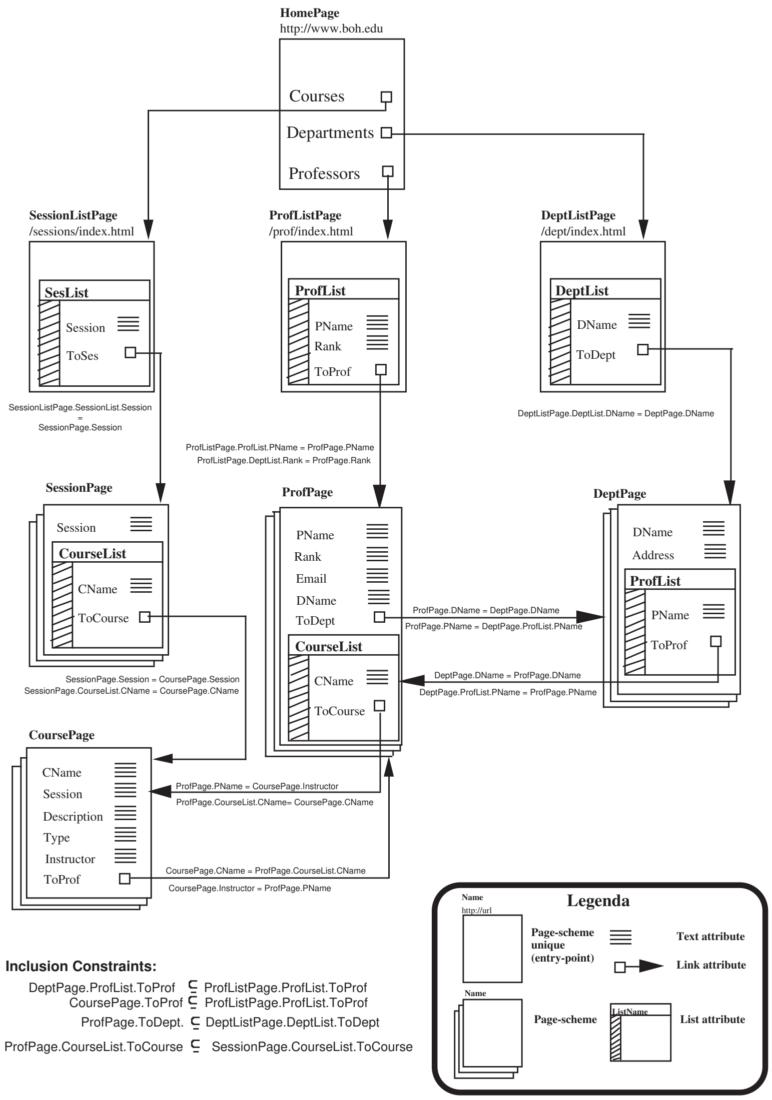

Fig. 1. The *Web-Scheme* of a University Web Site

exploited in query optimization, we enrich the model with two kinds of integrity constraints: *link constraints*, and *inclusion constraints*.

A *link constraint* is a predicate associated with a link. It is used to document the fact that the value of some attribute in the source page-relation equals the value of another attribute in a related tuple in the target page-relation. For example, with respect to Figure 1, this is the case for attribute `DName` in page-schemes `DeptPage` and `ProfPage` or for attribute `Session` in `SessionPage` and `CoursePage`. In our model, this can be documented by the following link constraints:

$$\texttt{ProfPage.DName = DeptPage.DName}$$
$$\texttt{SessionPage.Session = CoursePage.Session}$$

To formalize, given two page-schemes, P_1 and P_2 connected by a link ToP_2, a *link constraint* between P_1 and P_2 is any expression of the form: $A = B$, where A is a monovalued attribute of P_1 and B a monovalued attribute of P_2. Given an instance of the two page-schemes, we say that the link holds if: for each pair of tuples $t_1 \in P_1, t_2 \in P_2$, then attribute ToP_2 of t_1 equals attribute URL of t_2 if and only if attribute A of t_1 equals attribute B of t_2.

Besides link constraints, we also extend to the model the notion of *inclusion constraint*, in order to reason about containment among different navigation paths. Consider again Figure 1: it can be seen that page-scheme `ProfPage` can be reached either from `ProfListPage` or from `DeptPage` or from `CoursePage`. Since page-scheme `ProfListPage` corresponds to the list of all professors, it is easy to see that the following inclusion constraints hold:

$$\texttt{CoursePage.ToProf} \subseteq \texttt{ProfListPage.ProfList.ToProf}$$
$$\texttt{DeptPage.ProfList.ToProf} \subseteq \texttt{ProfListPage.ProfList.ToProf}$$

Note that the inverse containments do not hold in general. For example, following the path that goes through course pages, only professors that teach at least one course can be reached; but there may be professors who do not teach any courses.

To formalize, given a page-scheme P, and two link attributes L_1, L_2 towards P in P_1 and P_2, an *inclusion constraint* is an expression of the form: $P_1.L_1 \subseteq P_2.L_2$. Given instances p_1 and p_2 of each page-scheme, we say that the constraint holds if: for each tuple $t_1 \in p_1$, there is a tuple $t_2 \in p_2$, such that the value of L_1 in t_1 equals the value of L_2 in t_2. Two constraints of the form $P_1.L_1 \subseteq P_2.L_2$, $P_2.L_2 \subseteq P_1.L_1$ may be written in compact form as $P_1.L_1 \equiv P_2.L_2$.

Figure 1 also shows link and inclusion constraints for the Department example.

4 Navigational Algebra

In this Section we introduce the Navigational Algebra (nalg), an algebra for nested relations extended with navigational primitives. nalg is an abstraction of the practical language Ulixes [4] and is also similar in expressive power to (a subset of) *WebOQL* [2], and it allows the expression of queries against an adm scheme.

Besides the traditional selection, projection and join operators, in nalg two simple operators are introduced in order to describe navigation. The first operator, called *unnest page* is the traditional *unnest* [17] operator (μ), that allows to access data at different levels of nesting inside a page; instead of the traditional prefix notation: $\mu_A(R)$, in this paper we prefer to use a different symbol, $\diamond$, and an infix notation: $R \diamond A$. The second, called *follow link* and denoted by symbol $\longrightarrow$, is used to follow links. In some sense, we may say that $\diamond$ is used to navigate *inside* pages, i.e., inside the hierarchical structure of a page, whereas $\longrightarrow$ to navigate *outside*, i.e., between pages.

Note that the the selection-projection-join algebra is a sublanguage of our navigational algebra. In this way, we are able to manipulate both relational and navigational queries, as it is appropriate in the Web framework. To give an example, consider Figure 1. Suppose we are interested in the name and e-mail of all professors in the *Computer Science* department. To reach data of interest, we first need to navigate the site as follows:

$$\texttt{ProfListPage} \diamond \texttt{ProfList} \stackrel{ToProf}{\longrightarrow} \texttt{ProfPage}$$

The semantics of this expression is as follows: entry point `ProfListPage` is accessed through its URL; the corresponding nested relation is *unnested* with respect to attribute `ProfList` in order to be able to access attribute `ToProf`; finally, each of these links is followed to reach the corresponding `ProfPage`. Operator $\stackrel{ToProf}{\longrightarrow}$ essentially "expands" the source relation by joining it with the target one; the join is a particular one: since it physically corresponds to following links, it implicitly imposes the equality of the link attribute in the source relation with the URL attribute in the target one. We assume that attributes are suitably renamed whenever needed.

Since the result of the expression above is a (nested) relation containing a tuple for each tuple in page scheme `ProfPage`, the query *"Name and e-mail of all professors in the Computer Science Department"* can be expressed as follows:

$$\pi_{PName, e-mail}(\sigma_{DName='C.S.'}(\texttt{ProfListPage}\diamond\texttt{ProfList}\stackrel{ToProf}{\longrightarrow}\texttt{ProfPage})) \quad (1)$$

To formalize, the NAVIGATIONAL ALGEBRA is an algebra for the ADM model. The *operators* of the navigational algebra work on page-relations and return page-relations, as follows:

- *selection*, σ, *projection*, π and *join*, $\bowtie$, have the usual semantics;
- *unnest page*, $\diamond$, is a binary operator that takes as input a nested relation R and a nested attribute A of R; its semantics is defined as the result of unnesting R with respect to A: $R \diamond A = \mu_A(R)$.
- *follow link*, $\stackrel{L}{\longrightarrow}$, is a binary operator that takes as input two page-relations, R_1, R_2, such that there is a link attribute, L from R_1 to R_2; the execution of expression $R_1 \stackrel{L}{\longrightarrow} R_2$ corresponds to computing the join of R_1 and R_2 based on the link attribute, that is: $R_1 \stackrel{L}{\longrightarrow} R_2 = R_1 \bowtie_{R_1.L=R_2.URL} R_2$. It thus "expands" the source relation following links corresponding to attribute L.

A NALG *expression* over a scheme S is any combination of operators over page-relations in S. With each expression it is possible to associate in the usual way a *query tree* (or *query plan*) in which leaf nodes correspond to page-relations and all other nodes to NALG operators (see Figures 2, 3).

Note that not all navigational algebra expressions are computable. In fact, the only page-relations in a Web scheme that are directly accessible are the ones corresponding to entry-points, whose URL is known and documented in the scheme; thus, in order to be computable, all navigational paths involved in a query must start from an entry point. We thus define the notion of *computable expression* as a navigational algebra expression such that all leaf nodes in the corresponding query plan are entry points.

5 Querying Virtual Views of the Web

Our approach to querying the Web consists in offering a relational view of data in a portion of the Web, and allowing users to pose queries against this view. In this paper we concentrate on *conjunctive queries*. When a query is issued to the system, the query engine transparently navigates the Web and returns the answer. We assume that the query engine has knowledge about the following elements: (*i*) the ADM scheme of the site; (*ii*) the set of relations offered as external view to the user; we call these relations *external relations*; (*iii*) for each external relation, one or more computable navigational algebra expression whose execution correspond to materializing the extent of that external relation. Note that the use of both ADM and the navigational algebra is completely transparent to the user, whose perception of the query process relies only on the relational view and the relational query language.

To give an example, suppose we consider the Department site whose scheme is reported in Figure 1. Suppose also we are interested in pieces of information about Departments, Professors, and Courses. We may decide to offer a view of the site based on the following external relations:

1. `Dept(DName, Address)`;
2. `Professor(PName, Rank, email)`;
3. `ProfDept(PName, DName)`;
4. `Course(CName, Session, Description, Type)`;
5. `CourseInstructor(CName, PName)`;

In this case, in order to answer queries, the query engine must know the Web scheme in Figure 1 and the external scheme of items 1–5; moreover, it must also know how to navigate the scheme in order to build the extent of each external relation; this corresponds to associating with each external relation one or more computable NALG expressions, whose execution materializes the given relation. For example, with respect to external relations `Dept`, `Professor` and `ProfDept` above, we have the following navigations:

1. `Dept(DName, Address)`
   $= \pi_{DName,Address}(\texttt{DeptListPage}\diamond\texttt{DeptList} \stackrel{ToDept}{\longrightarrow} \texttt{DeptPage})$

2. `Professor(PName, Rank, email)`
   $= \pi_{PName,Rank,email}(\texttt{ProfListPage}\diamond\texttt{ProfList} \stackrel{ToProf}{\longrightarrow} \texttt{ProfPage})$

3. `ProfDept(PName, DName)`
   $= \pi_{PName,DName}(\texttt{ProfListPage}\diamond\texttt{ProfList} \stackrel{ToProf}{\longrightarrow} \texttt{ProfPage})$
   $= \pi_{PName,DName}(\texttt{DeptListPage}\diamond\texttt{DeptList} \stackrel{ToDept}{\longrightarrow} \texttt{DeptPage}\diamond\texttt{ProfList})$

We call these expressions the *default navigations* associated with external relations. There may be different alternative expressions associated with the same external relation (see 3). Note also that, for a given external relation, there may be other possible navigational expressions, "contained" in the default navigations. For example, professors may be reached also through their courses. However, it is not guaranteed that *all* professors may be reached using this path.

6 Query Optimization

When the system receives a query on the external view, it has to choose an efficient strategy to navigate the site and answer the query. The optimization proceeds as follows:

- the original query is translated into the corresponding projection-selection-join algebraic expression;
- this expression is converted into a computable NALG expression, which is repeatedly rewritten by applying NALG rewriting rules in order to derive a number of candidate execution plans, i.e., executable algebra expressions;
- finally, the cost of these alternatives is evaluated, and the best one is chosen, based on a specific cost model.

Since network accesses are considerably more expensive than memory accesses, we decide to adopt a simple *cost model* [10] based on the number of pages downloaded from the network. Thus, we aim at finding an execution plan for the query that minimizes the number of pages visited during the navigation. Note that the cost model can be made more accurate by taking into account also other parameters such as the size of pages, the deployment of Web servers over the network or the query *locality* [11]. Also, some expensive local operations should be considered. We omit these details here for the sake of simplicity. In the following section, we introduce a number of rewriting rules for the navigational algebra that can be used to this end. In [10] we develop an optimization algorithm based on these rules that, by successive rewritings, generates a number of candidate execution plans. Each of these is then evaluated based on the cost function, and the optimal one is chosen.

6.1 NALG Rewriting Rules

The first, fundamental rule simply says that, in order to evaluate a query that involves external relations, each external relation must be replaced by one of the corresponding NALG expressions. In fact, the extent of an external relation is not directly accessible, and must be built up by navigating the site.

Rule 1 [Default Navigation] *Each external relation can be replaced by any of its default navigations.*

Other rules are based on simple properties of the navigational algebra, and thus are rather straightforward, as follows.

Rule 2 *Given two relations R_1, R_2, such that R_1 has an attribute L of type* LINK TO *R_2, suppose that a link constraint $R_1.A = R_2.B$ is associated with L; then:* $R_1 \bowtie_{R_1.A=R_2.B} R_2 \; = \; R_1 \bowtie_{R_1.L=R_2.URL} R_2 \; = \; R_1 \xrightarrow{L} R_2$

Rule 3 *Given a relation R, suppose X is a set of non-nested attributes of R and A a nested attribute; then:* $\pi_X(R \diamond A) \; = \; \pi_X(R)$

Rule 4 *Given a relation R, suppose A is a nested attribute of R, and Y any set of non-nested attributes of R; then:* $(i) R \bowtie_Y R \; = \; R$; $(ii) (R \diamond A) \bowtie_Y R \; = \; R \diamond A$

Rule 5 *Given two relations R_1, R_2, suppose X is a set of attributes of R_1; suppose also that R_1 has an attribute L of type* LINK TO *R_2; then:*
$\pi_X(R_1 \xrightarrow{L} R_2) \;=\; \pi_X(R_1)$

The following two rules extend ordinary selection and projection pushing to navigations. They show how, based on link constraints, selections and projections can be moved down along a path, in order to reduce the size of intermediate results, and thus network accesses.

Rule 6 [Pushing Selections] *Given two relations R_1, R_2, such that R_1 has an attribute L of type* LINK TO *R_2, suppose that a link constraint $R_1.A = R_2.B$ is associated with L; then: $\sigma_{B='v'}(R_1 \xrightarrow{L} R_2) \;=\; \sigma_{A='v'}(R_1) \xrightarrow{L} R_2$*

Rule 7 [Pushing Projections] *Given two relations R_1, R_2, such that there is an attribute L in R_1 of type* LINK TO *R_2, suppose that a link constraint $R_1.A = R_2.B$ is associated with L; then: $\pi_B(R_1 \xrightarrow{L} R_2) \;=\; \pi_A(\pi_{A,L}(R_1) \xrightarrow{L} R_2)$*

We now concentrate on investigating the relationship between joins and navigations. The rules make use of link and inclusion constraints. The first rule (rule 8) states that, in all cases in which it is necessary to join the result of two different paths (denoted by R_1 and R_2) both pointing to R_3, it is possible to join the two sets of pointers in R_1 and R_2 *before* actually navigating to R_3, and *then* navigate the result.

Rule 8 [Pointer Join] *Given relations R_1, R_2 and R_3, such that both R_1 and R_2 have an attribute L of type* LINK TO *R_3, suppose that a link constraint $R_2.A = R_3.B$ is associated with L; then:*
$(R_1 \xrightarrow{L} R_3) \bowtie_{R_3.B=R_2.A} R_2 \;=\; (R_1 \bowtie_{R_1.L=R_2.L} R_2) \xrightarrow{L} R_3$

The second rule says that, in some cases, joins between page sets can be eliminated in favor of navigations; in essence, the join is implicitly computed by chasing links between pages.

Rule 9 [Pointer Chase] *Given relations R_1, R_2 and R_3, such that both R_1 and R_2 have an attribute L of type* LINK TO *R_3; suppose X is a set of attributes not belonging to R_1; suppose also that a link constraint $R_2.A = R_3.B$ is associated with L, and that there is an inclusion constraint $R_2.L \subseteq R_1.L$; then: $\pi_X((R_1 \xrightarrow{L} R_3) \bowtie_{R_3.B=R_2.A} R_2) \;=\; \pi_X(R_2 \xrightarrow{L} R_3)$*

7 Pointer-join vs Pointer-chase

Rules 8 and 9 essentially correspond to two alternative approaches to query optimization, which we have called the *"pointer join"* approach – aiming at reducing link traversal by pushing joins of link sets – versus a *"pointer chase"* approach – in which links between data are followed to restrict network access to relevant items. For a large number of queries, both strategies are possible. Our optimization algorithm is such that it generates and evaluates plans based on both strategies. In the following, we discuss this interaction between join and navigation in the Web and show that pointer chase is sometimes less expensive than joins.

Example 1. **[Pointer-Join]** Consider the scheme in Figure 1, and suppose we need to answer the following query: *"Name and Description of courses taught by full professors in the Fall session"*. The query can be expressed against the external view as follows:

$$\pi_{CName,Desc.}(\sigma_{Ses.=`Fall',Rank=`Full'}(\mathtt{Professor} \bowtie \mathtt{CourseInstructor} \bowtie \mathtt{Course}))$$

Note that there are several ways to rewrite the query. For example, since external relation `CourseInstructor` has two different default navigations, by rule 1, the very first rewrite step originates two different plans. Then, the number of plans increases due to the use of alternative rules. We examine only two of these possible rewritings, based on a pointer-join and a pointer-chase strategy, respectively, and discuss the relationship between the two.

The first rewriting is essentially based on rule 8, and corresponds to adopting a traditional optimization strategy, in which link chasing is reduced by using joins. The rewriting goes as follows:

$$\begin{aligned}
&\pi_{CName,Descr}(\sigma_{Ses.=`Fall',Rank=`Full'}(\mathtt{Professor} \bowtie_{PName} \mathtt{CourseInstructor}\\
&\qquad\qquad \bowtie_{CName} \mathtt{Course}))\\
&\overset{rule\ 1}{=} (1a)\ \pi_{CName,Descr}(\ \sigma_{Ses.=`Fall',Rank=`Full'}((\mathtt{ProfListPage}\diamond\mathtt{ProfList} \xrightarrow{ToProf} \mathtt{ProfPage})\\
&\qquad \bowtie_{PName} (\mathtt{ProfListPage}\diamond\mathtt{ProfList} \xrightarrow{ToProf} \mathtt{ProfPage}\diamond\mathtt{CourseList})\\
&\qquad \bowtie_{CName} (\mathtt{SessionListPage}\diamond\mathtt{SesList} \xrightarrow{ToSes} \mathtt{SessionPage}\diamond\mathtt{CourseList}\\
&\qquad\qquad \xrightarrow{ToCourse} \mathtt{CoursePage})))\\
&\overset{rule\ 4}{=} (1b)\ \pi_{CName,Descr}(\sigma_{Ses.=`Fall',Rank=`Full'}((\mathtt{ProfListPage}\diamond\mathtt{ProfList} \xrightarrow{ToProf} \mathtt{ProfPage}\diamond\mathtt{CourseList})\\
&\qquad \bowtie_{CName} (\mathtt{SessionListPage}\diamond\mathtt{SesList} \xrightarrow{ToSes}\\
&\qquad\qquad \mathtt{SessionPage}\diamond\mathtt{CourseList} \xrightarrow{ToCourse} \mathtt{CoursePage})))\\
&\overset{rule\ 8}{=} (1c)\ \pi_{CName,Descr}(\sigma_{Ses.=`Fall',Rank=`Full'}(((\mathtt{ProfListPage}\diamond\mathtt{ProfList} \xrightarrow{ToProf} \mathtt{ProfPage}\diamond\mathtt{CourseList})\\
&\qquad \bowtie_{ToCourse} (\mathtt{SessionListPage}\diamond\mathtt{SesList} \xrightarrow{ToSes} \mathtt{SessionPage}\diamond\mathtt{CourseList}))\\
&\qquad\qquad \xrightarrow{ToCourse} \mathtt{CoursePage}))\\
&\overset{rule\ 6}{=} (1d)\ \pi_{CName,Descr}(((\sigma_{Rank=`Full'}(\mathtt{ProfListPage}\diamond\mathtt{ProfList}) \xrightarrow{ToProf} \mathtt{ProfPage}\diamond\mathtt{CourseList})\\
&\qquad \bowtie_{ToCourse} (\sigma_{Ses.=`Fall'}(\mathtt{SessionListPage}\diamond\mathtt{SesList}) \xrightarrow{ToSes} \mathtt{SessionPage}\diamond\mathtt{CourseList}))\\
&\qquad\qquad \xrightarrow{ToCourse} \mathtt{CoursePage})
\end{aligned}$$

First, by rule 1, each external relation is replaced by a corresponding default navigation $(1a)$; then, rule 4 is applied to eliminate repeated navigations $(1b)$; then, by rule 8, the join is pushed down the query plan: in order to reduce the number of courses to navigate, we join the two pointer sets in `CourseList`, and then navigate link `ToCourse` $(1c)$; finally, based on link constraints, rule 6 is used to push selections down $(1d)$. The plan can then be further rewritten to push down projections as well.

A radically different way of rewriting the query is based on rule 9; in this case, the first two rewritings are the same as above; then, by rule 9, the join is removed in favor of navigations in the site ; finally, projections are pushed down to generate plan $(2d)$, as follows:

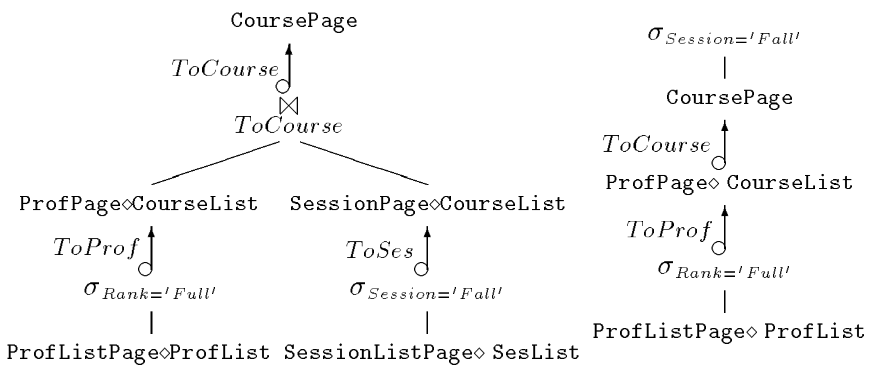

Fig. 2. Alternative Plans for the query in Example 1

$$(2d)\ \pi_{CName,Descr}(\sigma_{Ses.=`Fall'}(\sigma_{Rank=`Full'}(\texttt{ProfListPage}\diamond\texttt{ProfList}) \xrightarrow{ToProf} \texttt{ProfPage}\diamond\texttt{CourseList} \xrightarrow{ToCourse} \texttt{CoursePage}))$$

Plans corresponding to expressions $(1d)$ and $(2d)$ are represented in Figure 2. Plan $(1d)$ corresponds to: (i) finding all links to courses taught by full professors; (ii) finding all links to courses taught in the fall session; (iii) joining the two sets in order to obtain the intersection; (iv) navigate to the pages in the result. On the other hand, plan $(2d)$ corresponds to: (i) finding all full professors; (ii) navigating all courses taught by full professors; (iii) selecting courses in the fall section.

It is rather easy to see that plan $(1d)$ has a lower cost. In fact, plan $(2d)$ navigates *all* courses taught by full professors, and then selects the ones belonging to the result; on the contrary, in plan $(1d)$, pointers to courses are first selected, and then only pages belonging to the result are navigated.

The pointer-join strategy chosen by the optimizer in Example 1 is reminiscent of the ones that have been proposed for relational databases to optimize selections on a relation with multiple indices [12], and for object-oriented query processors to reduce pointer chasing in evaluating path-expressions – assuming a join index on professors and courses is available [8].

However, the following two examples show that, in the Web context, this is not always the optimal solution: in some cases, pointer-chasing is less expensive. This is shown in the following example.

Example 2. **[Pointer-chasing]** Consider the scheme in Figure 1, and suppose we need to answer the following query: *"Name and Email of Professors who are members of the Computer Science Department, and who are instructors of Graduate Courses"*. The query can be expressed on the external view as follows:

$$\pi_{PName,Email}(\sigma_{DName=`C.S.',Type='Graduate'}(\texttt{Course} \bowtie \texttt{CourseInstructor} \bowtie \texttt{Professor} \bowtie \texttt{ProfDept}))$$

We examine the two most interesting candidate execution plans. A pointer-join approach yields an expression (1), in which rule 8 is applied to join links `ToProf`

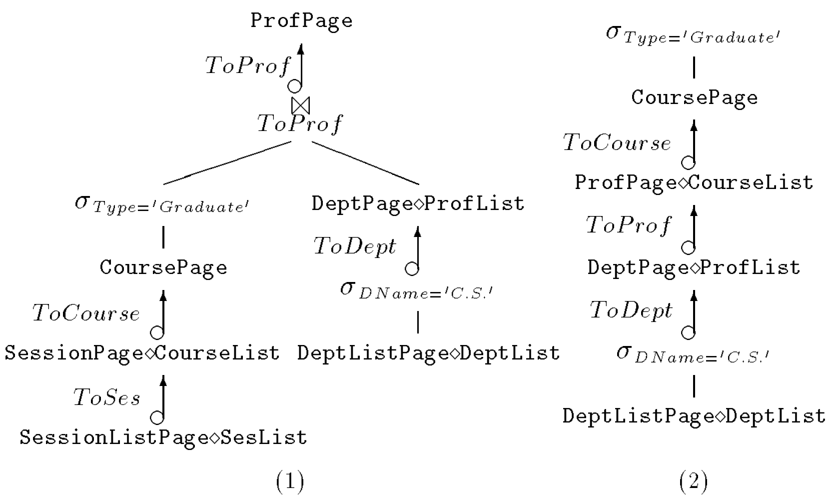

Fig. 3. Alternative Plans for the query in Example 2

in `CoursePage` and `ProfList` before navigating to `ProfPage`; then, rule 6 is used to push down selections. The alternative pointer-chasing strategy (2) corresponds to completely eliminate joins by replacing them with navigations. The query plans corresponding to these expressions are in Figure 3.

Let us compare the cost of the two plans. Plan (1) intersects two pointer sets, obtained as follows: the left-hand side path navigates the Computer Science Department page and retrieves all pointers to its members; the right-hand side path essentially downloads all session pages, and all course pages, and derives all pointers to instructors of graduate courses; then, the two pointer sets are joined, and URLs are navigated to build the result. On the contrary, plan (2) downloads all pages of professors in the Computer Science Department, and, from those, the pages of the corresponding courses. Now, a little reflection shows that plan (2) has a lower cost: navigating all instances of `CoursePage` in plan (1) makes it excessively expensive. In fact, due to the topology of the site, we know that there are several professors for each Department and several courses for each professor.

An intuitive explanation of this fact is the following: in this case, there is no efficient access structure to page-scheme `CoursePage`: in order to select graduate courses, it is necessary to navigate all courses; this makes the cost excessively high, and the pointer-join approach fails. On the contrary, following links from the Computer Science Department yields a reasonable degree of selectivity, that reduces the number of network accesses.

Based on the previous examples, we can conclude that ordinary pointer-join techniques do not transfer directly to the Web; a number of new issues have to be taken into account, namely, the different cost model and the absence of adequate access structures; in general, several alternative strategies, based on pointer-chasing, need to be evaluated.

Acknowledgments The authors would like to thank Paolo Atzeni and Giuseppe Sindoni, for useful discussions on early drafts of this paper. Special thanks go to Alessandro

Masci, who implemented the navigational algebra and the relational view manager, provided insightful comments and supported us in every phase of this work. This work was in part done while the third author was visiting the University of Toronto. The first and the third author were partially supported by Università di Roma Tre, MURST, and Consiglio Nazionale delle Ricerche. The second author was supported by the Natural Sciences and Engineering Research Council of Canada and the Center for Information Technology of Ontario.

References

1. S. Abiteboul and V. Vianu. Regular path queries with constraints. In *PODS'97*.
2. G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring documents, databases and Webs. In *ICDE'98*, 1998.
3. P. Atzeni and G. Mecca. Cut and Paste. In *PODS'97*.
4. P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *VLDB'97*.
5. C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *ICDT'90*.
6. S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *SIGMOD'82*.
7. B. P. Jenq, D. Woelk, W. Kim, and W. Lee. Query processing in distributed ORION. In *EDBT'90*.
8. A. Kemper and G. Moerkotte. Access support relations: An indexing method for object bases. *Information Systems*, 17(2):117–145, 1992.
9. M. Ley. Database systems and logic programming bibliography site. `http://-www.informatik.uni-trier.de/~ley/db/index.html`.
10. G. Mecca, A. Mendelzon, and P. Merialdo. Efficient queries over Web views. Technical Report n. RT-DIA-31-1998, Dipartimento di Informatica e Automazione, Università di Roma Tre, 1998. `http://poincare.dia.uniroma3.it:8080/Araneus/`.
11. A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. *Journal of Digital Libraries*, 1(1):54–67, April 1997.
12. C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *EDBT'90*.
13. S. Navathe. An intuitive view to normalize network structured data. In *VLDB'80*.
14. M. T. Özsu and J. A. Blakeley. Query processing in object-oriented database systems. In W. Kim, editor, *Modern Database Management – Object-Oriented and Multidatabase Technologies*, pages 146–174. Addison Wesley-ACM Press, 1994.
15. A. Rosenthal and D. S. Reiner. An architecture for query optimization. In *SIGMOD*, 1982.
16. A. Rosenthal and D. S. Reiner. Querying relational views of networks. In W. Kim, D. S. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*, pages 109–124. Springer-Verlag, 1985.
17. M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for ¬1NF relational databases. *ACM TODS*, 13(4):389–417, December 1988.
18. G. M. Shaw and S. B. Zdonik. An object-oriented query algebra. In *DBPL'89*.
19. P. Valduriez. Join indices. *ACM TODS*, 12(2):218–246, 1987.
20. Xie Z. and Han J. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *VLDB'94*.
21. C. Zaniolo. Design of relational views over network schemas. In *SIGMOD* 1979.
22. C. Zaniolo. The database language GEM. In *SIGMOD* 1983.

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