Distributed Query Evaluation on Semistructured Data*

Dan Suciu†
University of Washington

Abstract

Semistructured data is modeled as a rooted, labeled graph. The simplest kinds of queries on such data are those which traverse paths described by regular path expressions. More complex queries combine several regular path expressions, with complex data restructuring, and with sub-queries. This paper addresses the problem of efficient query evaluation on distributed, semistructured databases. In our setting the nodes of the database are distributed over a fixed number of sites, and the edges are classified into local (with both ends in the same site) and cross edges (with ends in two distinct sites). *Efficient evaluation* in this context means that the number of communication steps is fixed (independent on the data or the query), and that the total amount of data sent depends only on the number of cross links and of the size of the query's result. We give such algorithms in three different settings. First, for the simple case of queries consisting of a single regular expression. Second, for all queries in a calculus for graphs based on *structural recursion* which in addition to regular path expressions can perform non-trivial restructuring of the graph. And third, for a class of queries we call *select-where queries* which combine pattern matching and regular path expressions with data restructuring and sub-queries. The paper also includes a discussion on how these methods can be used to derive efficient view maintenance algorithms.

Disclaimer This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.

1 Introduction

The semistructured data model has been introduced for data which does not conform to a rigid schema [Abi97, Bun97]. Data components can be missing in some items, can have different types in different items, collections can be heterogeneous and nested [QRS+95]. While originally motivated by a variety of applications such as integration of heterogeneous data sources [PGMW95], management of biological databases [TMD92, BDHS96], or querying or managing Web sites [MMM96, KS95, FFK+97], the dominant application today is for data sharing on the Web. XML [Con98], the standard data format for the Web, is essentially a syntax for semistructured data.

*Copyright 200x by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to Post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1(212)869 − 0481, or permissions@acm.org.

†Part of this work was done while the author was at AT&T Shannon Laboratories

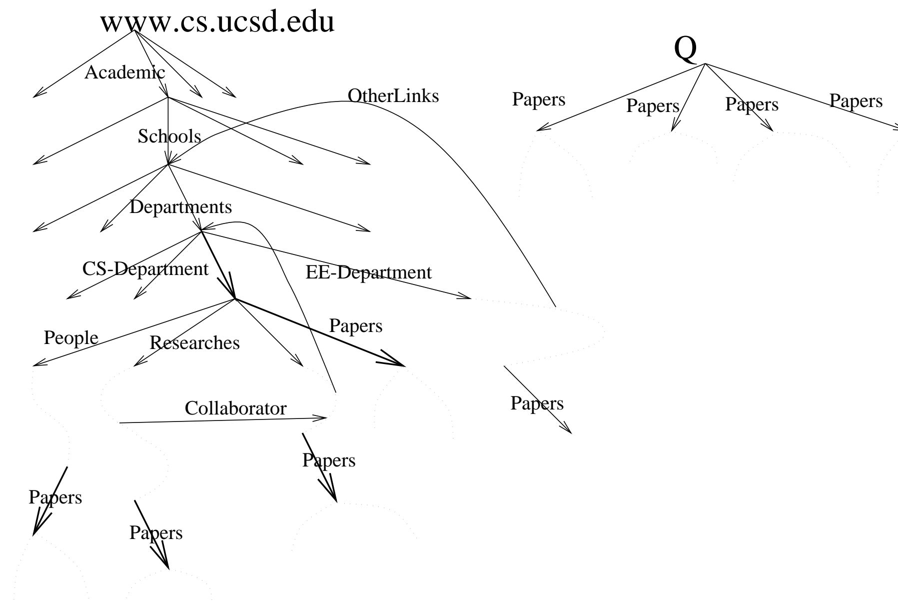


Figure 1: A fragment of the web site at `http://www.ucsd.edu`, and the result of a query Q. For the purpose of structure specific queries such a data source can be modeled as an edge labeled tree.

In the semistructured data model a database is modeled as a rooted, labeled graph. The nodes represent objects and have an associated oid. The labels are atomic values, like strings, integers, etc, or large objects, like images, sounds, etc, which are still atomic for the purpose of query evaluation. Labels can be attached to either the graphs nodes or edges or both, resulting in minor variations of the semistructured data model. For uniformity we assume here the labels to be attached to edges only. Web-sites are a good examples of semistructured data instances: Figure 1 illustrates an example of such a semistructured databases: a fragment of the Web site as `http://www.ucsd.edu`.

In data sharing applications the semistructured data is often distributed. Several autonomous sites hold local data that may include links to objects in other sites. For example, consider the database represented in Figure 2, which is distributed on two sites, `http://www.ucsd.edu` and `http://www.sdsc.edu` respectively. Each node now belongs to exactly one site, while the edges can be either local (at one site), or cross edges (between the two sites). Such cross edges are expressed in HTML as references to other Web pages and in XML as links to other XML elements. XLink [DMOT00] defines a framework for expressing such links in XML.

In this work we address the problem of efficient query evaluation on distributed, semistructured databases. The scenario we consider is one in which the data is distributed on a number of sites (say dozens or more, but not thousands) that have agreed to cooperate both in sharing data and in answering queries. While document sharing is common on the Web today, Web servers do not answer user queries. Currently a W3C working group is designing a query language for XML [CFMR00] that will likely lead in the future XML servers to accept queries on their XML data. Access to such servers will be restricted to authorized users, for example paying customers, business partners, or users inside a firewall. Our scenario for distributed query evaluation is one in which a community of XML data providers agree to share their data. *Sharing* is implemented by having links from the local XML data to XML objects at some other server in the community, for example by using XLink [DMOT00]. The community is loose, with the data providers being autonomous. In particular there are no global integrity conditions, for example broken links are possible. At the logical level users see a unique, global graph, and are unaware of the distribution.

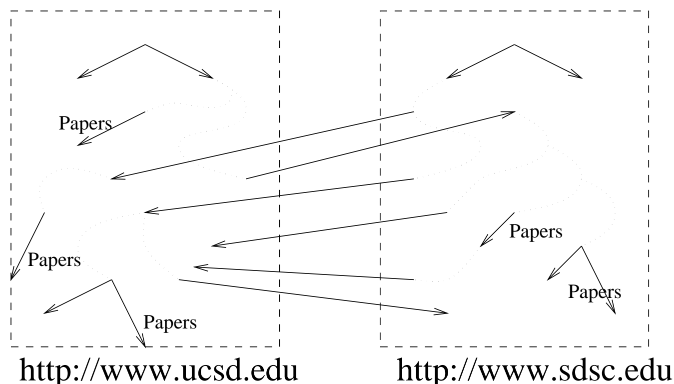


Figure 2: The information source is distributed on two sites.

Each query is initiated at some server, and may have to extract data from other servers in order to compute the answer. Each server will restrict access to authorized users and the other servers.

For example in a B2B application a parts supplier may offer its customers access to XML data about its products and allow the customer's applications to query that data using an XML query language. The supplier's data may contain links to other sites, for example to a shipping company for up to date information about delivery options, and it will have an agreement with that shipping company to support distributed evaluation of queries on their joint data.

One consequence of this scenario is that objects are fetched using the HTTP protocol, which makes the connection setup cost much higher than the data transfer cost. One single connection that transfers a lot of data is preferable to several smaller connections each transferring small amounts of data. A second consequence is the predominance of relative links over absolute links, in order to reduce the likelihood of broken links. An absolute link consists of the referred object's oid while a relative link consists of some object's oid plus a navigation expression describing how to reach from there the target object. XLink allows relative links, with the navigation being expressed in XPath [Cla99a]. Rather than referring to the oid of some specific object at a remote site, we may prefer to refer to a more important object that is less likely to change in time, then use a relative path expression to reach our real target. Thus only a small subset of the objects at a site are directly referred by other sites.

Several semistructured data and XML query languages have been proposed by researchers. At their core, all share a common feature: they allow navigation through the data with regular path expressions. For example the following UnQL [BDHS96, BFS00] query:

$$\begin{aligned} Q1 = &\ \mathsf{select}\ t \\ &\ \mathsf{where}\ * \Rightarrow \textit{"CS-Department"} \Rightarrow * \Rightarrow \textit{"Papers"} \Rightarrow t\ \mathsf{in}\ db \end{aligned}$$

retrieves all papers accessible from a *"CS-Department"* link in the database in Figure 1. Here $* \Rightarrow$ *"CS-Department"* $\Rightarrow * \Rightarrow$ *"Papers"* is a *regular path expression*, in this case denoting any path which has some edge labeled *"CS-Department"*, and later some other edge labeled *"Papers"*. More complex queries may pose more complex conditions, consisting of several regular path expressions, or may construct results which are themselves graphs.

This paper discusses distributed evaluation techniques for three language constructs found in current semistructured data query languages: regular path expressions, recursive traversals of the graphs, and patterns that include regular path expressions. Our goal is to describe algorithms with proven efficient upper bounds. For example, consider the distributed database in Figure 2.

A naive evaluation of the previous query would be to chase the regular expression along the links in a "spider" style: this would result in the query being shipped between the two sites back and forth a number of times dependent on the number of cross links. In some restricted cases we may be able to determine ahead of time that the spider needs to be shipped between sites only a very small number of times, but in the general case the upper bound of the spider algorithm has an unacceptably high cost.

The definition of an *efficient* distributed query evaluation that we adopt throughout the paper is the following:

Definition 1.1 *An evaluation algorithm on a distributed database is* **efficient** *iff:*

1. *The total number of communication steps is constant, i.e. independent on the data or on the query. A communication step can be a broadcast, or a gather, and can involve arbitrarily large messages (but see condition 2).*

2. *The total amount of data transfered during query evaluation should depend only on (a) the size of the query's answer, and (b) the total number of cross edges.*

This definition is motivated by our application scenario. Setting up an HTTP transfer is costly, hence the best theoretical upper bound we can hope for is a constant number, independent on the size of the data and the query. On the other hand transfer costs are low compared to setup cost, so large messages are tolerable. We also attempt to minimize the amount of data transfered, but there are some natural barriers to that, in a worst case scenario. We obviously need to transfer the entire query answer, and, in the worst case, we have to traverse all cross links: hence the upper bound in 2.

Considering Figure 2, the "spider" evaluation algorithm of a regular path expression query violates item 1 above. Another naive evaluation method is to send the entire database to a single site in one communication step, then compute the query locally: this violates 2, since the size of the database is in general larger than the number of cross edges or of the size of the query's result. On the other hand an efficient algorithm according to our definition is not necessarily the best in certain practical settings.

Point 2 of the definition refers to the algorithm's *data complexity* in the sense of [Var82], not the query complexity or combined complexity. This means that the query is fixed and we analyze an algorithm's complexity (in our case the total amount of data transfered) as a function of the size of the data and the number of cross links.

We address the distributed query evaluation problem in three overlapping frameworks. First we consider just regular expression queries, whose form is $\mathsf{select}\ t\ \mathsf{where}\ R \Rightarrow t\ \mathsf{in}\ db$, which selects all nodes in a graph reachable from the root via a given regular expression R. Efficient evaluation of such queries reduces to the efficient computation of transitive closure of a distributed graph. For that, we give a straightforward efficient algorithm. Parallel algorithms for the evaluation of transitive closure have been studied before [VK88], but our setting here is different, since we allow large blocks of communications, and consider each communication to be very expensive. Our framework also differs from the traditional framework for distributed algorithms on graphs, e.g. for the computation of transitive closure [Lyn97]: there each node of the graph is stored on a separate site, and the number of communication steps is the graph's diameter.

Second, we consider a larger class of queries, which allows us to perform graph restructurings. These queries are described in a formalism whose central construct is a form of *structural recursion on trees* [BDS95, BDHS96, BFS00][1]. This allows us to define a query as a collection of mutually

[1]We follow here the definition of structural recursion for semistructured data from in [BFS00], which ensures that

recursive functions which iterate on the graph's structure. The queries in this formalism form an algebra $\mathcal{C}$, which is a fragment of UnQL [BDHS96, BFS00]: $\mathcal{C}$ doesn't have joins and uses only structural recursion expressions without nested functions. Still, $\mathcal{C}$ can express complex graph restructuring queries. It is shown in [BFS00, ABS99] that XSLT [Cla99b], a language for XML transformations standardized by the W3C, relies on structural recursion as its evaluation model, and that a certain core fragment of XSLT corresponds to structural recursion. That fragment can also be expressed in the algebra $\mathcal{C}$ discussed in this paper.

We develop an algebraic approach to distributed query evaluation for queries in $\mathcal{C}$, rather than an operational one. Namely for each query Q we show how to construct a related query Q^{dec}, which we call a *decomposed* query, such that on a distributed database, Q can be evaluated by evaluating Q^{dec} independently at each site, computing the accessible part of all result fragments, then shipping and assembling the separate result fragments at the client: moreover this evaluation is efficient, according to Definition 1.1. An algebraic approach is more powerful than an procedural one: since Q^{dec} is still a query, further optimizations can be done at each site, possible tailored specifically to each site. For example each site may transform and optimize Q^{dec} based on its local schema information, e.g. using techniques in [FS98]. For the correctness of this algebraic method we rely on the algebraic machinery developed in [BFS00]: we believe that distributed query evaluation is a nice illustration of the power of that machinery.

Third, we return to declarative queries and consider queries which combine patterns with regular expressions, sub-queries, and certain forms of restructurings. We call such queries *select-where* queries. We follow here the definition from UnQL [BDHS96, BFS00], a language designed for the semistructured data model, but notice that such queries correspond to join-free fragments of the XML query languages XML-QL [DFF$^+$99] and Quilt [CFR00]. We describe an efficient distributed evaluation algorithm for all join-free select-where queries: here too "efficient" is in the sense of Definition 1.1. We use two novel ideas in this algorithm. The first is to show that every select-where query Q can be evaluated in two stages: evaluate a related query Q_r to get a *partial result* P, then send P to the client and restructure it there to get the final result. The key property here is that Q_r is always a query expressible in $\mathcal{C}$, hence we know that it can be efficiently computed in parallel. Still, the size of P may be much larger than the query result and sending all its fragments to the client may violate condition 2 of Definition 1.1. The second idea relates to the way P is trimmed before it is sent to the client. We show that the useful portion of P can be computed by solving an *Alternating Graph Accessibility Problem*, AGAP [Imm87, GHR95]. In AGAP we are given a graph whose nodes are partitioned into AND nodes and OR nodes, and we have to determine whether a given node is *accessible*. By definition an OR node is accessible if at least one of its successors is accessible, while an AND node is accessible if all its successors are accessible. In general the AGAP is more difficult to evaluate in parallel than GAP (*Graph Accessibility Problem*), because AGAP is PTIME complete while GAP is in NC [GHR95]. However we notice that computing P's accessible part requires solving a particular form of the AGAP, namely with a bounded *AND-outdegree* (defined formally in Subsection 6.3). We describe an efficient distributed algorithm for computing this particular instance of AGAP.

Finally, for all three frameworks we show how the efficient evaluation techniques can be applied to the incremental view maintenance problem [GL95], for views on semistructured databases. In this problem we are given a centralized database db and a materialized view V defined in terms of a query, i.e. $V = Q(db)$. We are required to compute the new view $V' = Q(db')$, when the database db is updated with an increment Δ to become db'. Moreover the amount of work performed should

all structural recursion expressions have PTIME data complexity. In an earlier work [BDS95] structural recursion was defined as a more powerful construct.

depend only on the size of the view and that of the increment Δ. Sometimes we are allowed to store some additional information besides V, which is only used for the purpose of incremental view maintenance. We restrict Δ to consists only of additions to db (i.e. no deletions). Then we derive view maintenance algorithms by instantiating the distributed evaluation algorithms to a database consisting of two fragments: db and Δ.

The paper is organized as follows. We revise the graph data model in Section 2, following [BDHS96, BFS00]. We define regular queries and give an efficient distributed evaluation algorithm in Section 3. In Section 4 we define select-where queries, and explain why their distributed evaluation is more difficult than for regular path expressions. We define structural recursion and the graph data model in Section 5, following [BFS00], then present the algebraic method for distributed evaluation of $\mathcal{C}$ queries. We describe efficient distributive evaluation of arbitrary select-where queries in Section 6. Finally we discuss view maintenance in Section 7, and conclude in Section 8.

The most important results in this paper are novel. Namely those in Sections 3 and 6 are novel, while those in Sections 5 and 7 are based on the preliminary work [Suc96].

2 Data Model

Semistructured data is modeled as labeled graphs. Some variations can be found in the literature, but they are minor [ABS99]. Given our focus on the language UnQL we follow here the definition in [BDS95, BDHS96, BFS00], which has two features not present in other data models: markers and ε-edges. These features turn out to be convenient in describing data distribution and distributed query evaluation.

2.1 Rooted, Labeled Graphs

Let *Label* be the universe of all atomic values:

$$Label \stackrel{\text{def}}{=} Int \cup Real \cup Bool \cup String \cup Image \ldots$$

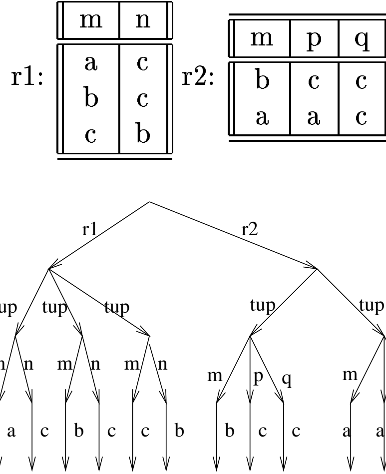
In our query language we can test the type of a given label a, with predicates like *isInt*(a), *isReal*(a), *isImage*(a), etc.

A semistructured database is a **rooted graph** (i.e. a graph with a distinguished node called the *root*), whose edges are **labeled** with elements from $Label \cup \{\varepsilon\}$ — hence the name "rooted, labeled graph". Here ε is a special label, denoting an "empty" symbol: we will discuss it in the sequel.

We sometimes call these graphs **trees**, since the main intuition underlying their associated operators comes from processing trees. However, unless explicitly mentioned, they are *graphs*, i.e. may have cycles and subgraph sharing. We denote with DB the set of all graphs.

Figure 1 contains an example of a semistructured data, namely a fragment of the web site `http://www.ucsd.edu`. Nodes in the graph correspond to web pages, while edges correspond to hyper-links. The assumption we make here is that all relevant information is on the edges. In reality, in the case of a web site, lots of information is stored at the nodes (i.e. in the web pages). The page content is not necessarily lost in our model, since, at the logical level, we can always move the information from the nodes into the incoming edges. For the purpose of keeping the query language simple, it is more convenient to assume that information is attached to edges only — hence *edge-labeled graphs*.

Figure 3 illustrates how semistructured data extends relational one. In this example a relational database with two relations, r_1, r_2 is represented as a tree. Notice that the data is self-describing:


r1:

m	n
a	c
b	c
c	b

r2:

m	p	q
b	c	c
a	a	c

Figure 3: A relational database represented as a tree.

for example the attribute names m, n, p, q are now part of the data, and not of the schema. One can easily generalize from this example and note that any relational database can be represented as a tree of depth 4.

In both examples all nodes in the graph are accessible from the root: any unaccessible parts of a rooted graph may be deleted. We will make this statement formal below.

These two examples represent the two extremes of semi-structured data: from unstructured (the web) to fully structured (relational data). In between there is a full spectrum of partially structured data which can be represented as labeled graphs, as argued convincingly in [QRS+95]: data with missing attributes, attributes which can be single- or set-valued, heterogeneous sets.

2.2 Syntax

Following [BDHS96, BFS00] we use a concrete syntax for denoting a particular case of semistructured data: data whose underlying graph is a tree. The syntax is:

$$T \quad ::= \quad \{\} \mid \{Label \Rightarrow T\} \mid T \cup T$$

We will abbreviate $\{a_1 \Rightarrow t_1, \ldots, a_n \Rightarrow t_n\}$ for $\{a_1 \Rightarrow t_1\} \cup \ldots \cup \{a_n \Rightarrow t_n\}$, and $\{a\}$ for $\{a \Rightarrow \{\}\}$. Then the example in Figure 3 is written as:

$$\begin{array}{l}
\{r1 \Rightarrow \{tup \Rightarrow \{m \Rightarrow \{a\}, n \Rightarrow \{c\}\}, \\
\qquad\qquad tup \Rightarrow \{m \Rightarrow \{b\}, n \Rightarrow \{c\}\}, \\
\qquad\qquad tup \Rightarrow \{m \Rightarrow \{c\}, n \Rightarrow \{b\}\}\}, \\
\;\; r2 \Rightarrow \{tup \Rightarrow \{m \Rightarrow \{b\}, p \Rightarrow \{c\}, q \Rightarrow \{c\}\}, \\
\qquad\qquad tup \Rightarrow \{m \Rightarrow \{a\}, p \Rightarrow \{a\}, q \Rightarrow \{c\}\}\}\}
\end{array}$$

Our data model has set semantics. E.g. the trees $\{a, b \Rightarrow \{c\}, b \Rightarrow \{c\}\}$ and $\{a, b \Rightarrow \{c\}\}$ are considered equal, and $t_1 \cup t_2$ should be read as set union. We will explain this in Sec. 2.4.

2.3 Epsilon Edges

As said earlier, we allow edges to be labeled with a special symbol, ε. The meaning of such an edge is related to that of an empty transition in automata [Aho90, pp.282], and is just a

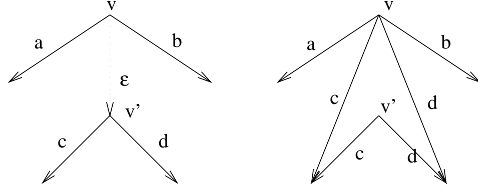


Figure 4: An ε edge from node v to v' means that all edges "visible" from v' should be "visible" from v too: the second tree is equivalent to the first one. Note that, if no other edge points to v', then v' may be eliminated, since it is no longer accessible from the root.

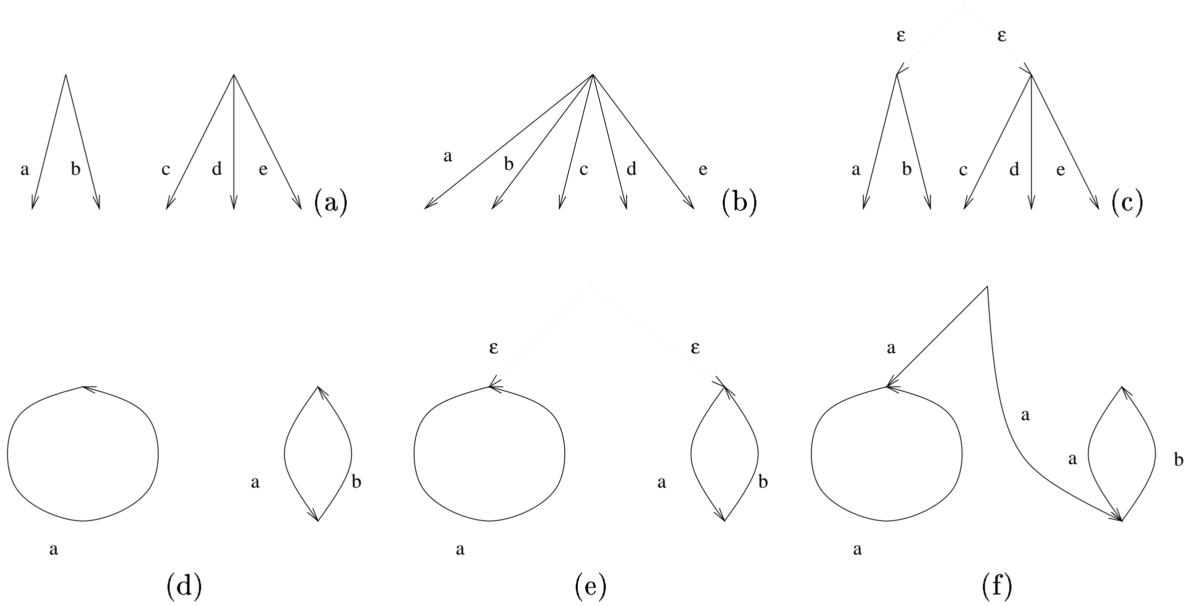


Figure 5: Illustration of how ε-edges can be used to express union.

notational convenience for describing succinctly more complex graphs. Whenever two vertices v, v' are connected by an ε edge, the intended meaning is that all edges emerging from v' should also emerge from v. This is illustrated in Figure 4.

Figure 5 illustrates why ε-edges are a convenient tool for representing graphs. Considering the two trees t_1, t_2 of Figure 5 (a), their union can be represented either as in (b) or as in (c): of course, (c) is simpler to explain and construct than (b), since the latter involves "merging" of two nodes, which raises the question what to do with their incoming edges. To see that more clearly, consider the case when t_1, t_2 are with graphs with cycles, like in Figure 5 (d). Then the meaning of $t_1 \cup t_2$ is best understood when described with ε-edges, as in (e). At a later step the ε-edges can be eliminated, to obtain the graph in (f).

2.4 Equality and Bisimulation

According to our set semantics the semi-structured data $\{a, b \Rightarrow \{c\}, b \Rightarrow \{c\}\}$ and $\{a, b \Rightarrow \{c\}\}$ are "equal". However they correspond to two different graphs: what does equality mean for graphs ? Following [BDS95, BDHS96, BFS00] we say that two graphs are equal if they are *bisimilar*. The

definition of bisimulation is rather technical, but it has a simple intuitive explanation. We give the technical definition first. Given two nodes u, v in some graph G, we write $u \xrightarrow{\varepsilon^*} \xrightarrow{a} v$ when there exists a path from u to v of length ≥ 1 in which the last edge is labeled $a \in \mathit{Label}$ and all previous edges are labeled ε.

Definition 2.1 *Given two rooted, labeled graphs G, G', a* **bisimulation** *from G to G' is a binary relation $R \subseteq \mathit{Nodes}(G) \times \mathit{Nodes}(G')$ such that:*

1. *$(\mathit{Root}(G), \mathit{Root}(G')) \in R$,*
2. *whenever $(u, u') \in R$ and there exists a path $u \xrightarrow{\varepsilon^*} \xrightarrow{a} v$ in G, then there exists some path $u' \xrightarrow{\varepsilon^*} \xrightarrow{a} v'$ (of possible different length, but with the same last label), and $(v, v') \in R$.*
3. *Similarly, but with the roles of G, G' reversed: whenever $(u, u') \in R$ and there exists a path $u' \xrightarrow{\varepsilon^*} \xrightarrow{a} v'$ in G', then there exists some path $u \xrightarrow{\varepsilon^*} \xrightarrow{a} v$, and $(v, v') \in R$.*

G, G' are *bisimilar* if there exists a bisimulation from G to G'.

The definition here of a bisimulation differs in some subtle way from that of a weak bisimulation in process algebra, as defined in [Mil89]. We refer the reader to [BFS00] for a discussion.

This rather technical definition admits a simple intuitive rephrasing. To test whether two graphs G and G' are bisimilar check if they become equal after (1) unfolding into (possible infinite) trees, (2) eliminating ε-edges like suggested in Fig. 4, and (3) performing duplicate elimination at each node.

We adopt bisimulation as our notion of graph equality. This has three important consequences. First, on encodings of sets bisimulation coincides with set equality. For example the graphs corresponding to $\{a, b \Rightarrow \{c\}, b \Rightarrow \{c\}\}$ and $\{a, b \Rightarrow \{c\}\}$ are bisimilar. Second, bisimulation is the formal underpinning of ε-edge elimination: that is, after eliminating a ε-edge like in Figure 4 the resulting graph is bisimilar to the original one. Third, every rooted graph is bisimilar to its accessible part: hence we can eliminate the unaccessible part from any graph and obtain a graph that is "equal" to it.

2.5 OIDs and Bag Semantics

Most other models of semistructured data considered in the literature have oids associated to nodes and allow the query language to check equality of node oids. The lack of oids is not a limitation of our model. Oid's can be always represented as data values, and hence can be tested for equality in any query language that allows equality tests on atomic values. We prefer here a model without oid's because simple set theoretic operations (such as union, $t \cup t'$) are easier to define, and this makes query languages performing complex restructuring easier to formalize. Languages based on oid's can also express such restructurings with the aid of Skolem functions. The choice between having oids or not is ultimately a matter of taste.

A related issue is that of bag v.s. set semantics. Our data model assumes set semantics, hence duplicates are eliminated from any collection. In practice one often needs bag semantics, for example in queries with aggregation (which we did not include in our language). Bag semantics is easy to model in a language with set semantics, by using oids: simply add to each element of a collection some distinct oid that prevents duplicates from being eliminated. For example, suppose a query retrieves all colors from a collection of products. If there are one thousand different products, but only (say) ten different colors, then in our data model the result set will have only ten elements, since duplicates have been eliminated. If the user wants to keep duplicates, then the query can

be reformulated to return a collection of (color, product-oid) pairs. Since the product oids for all one thousand products are different, there are no duplicates in the set, and duplicate colors will be preserved.

3 Queries with Regular Path Expressions

We describe in this section the simplest kind of queries on semi-structured data: regular path expression queries. For these we give a simple distributed evaluation algorithm, which is *efficient* in the sense of Definition 1.1.

3.1 Regular Path Expressions

The syntax for regular expressions is:

$$R ::= P \mid a \mid _ \mid R|R \mid R \Rightarrow R \mid R* \mid (R)$$

Here P is any unary, user-defined predicate on *Label*, or boolean combinations of such predicates, and $a \in Label$ is any label constant[2]. The expression _ denotes any label[3], $R_1|R_2$ denotes alternation, $R_1 \Rightarrow R_2$ concatenation, and $R*$ the Kleene closure.

Example 3.1 The following path regular expression finds all papers in the Computer Science Department:

$$_* \Rightarrow CSDept \Rightarrow _* \Rightarrow Paper$$

Here we assume *CSDept* and *Paper* to be user-defined predicates which recognizes strings denoting a Computer Science Department or a Paper respectively. For example we may define $Paper(x) \stackrel{\text{def}}{=} (x = \text{“}Publication\text{”} \text{ or } x = \text{“}Paper\text{”} \text{ or } x = \text{“}Technical\ Report\text{”})$.

However, the query may return papers from other departments, if some node under the Computer Science link has a link to some other department. To avoid that we may want to use a more complex regular expression:

$$_* \Rightarrow CSDept \Rightarrow (not(Dept)) * \Rightarrow Paper$$

This matches any path whose sequence of labels $a_1 a_2 \ldots a_n$ satisfies:

$$\exists m.1 \leq m < n \wedge CSDept(a_m) \wedge (\forall i = m+1, \ldots, n-1.\ not(Dept(a_i))) \wedge Paper(a_n)$$

□

Whenever no confusion arises, we will abbreviate $_*$ with $*$. Thus, the former expression becomes: $* \Rightarrow CSDept \Rightarrow (not(Dept)) * \Rightarrow Paper$.

We use regular expressions in queries of the form:

$$Q(db) = \text{select } t \quad \text{where } R \Rightarrow t \text{ in } db$$

We call such a query a *regular path expression query*, or *regular query* in short. Here t is a variable, which we call *tree variable*.

[2]Thus a is the same as the predicate P, where $P(x) = \text{if } x = a \text{ then } true \text{ else } false$

[3]Thus _ is the same as the predicate *true*.

Intuitively a regular query retrieves all nodes in db accessible from the root through a path whose labels match R. To be precise, we define the meaning as follows. Let $u_1, \ldots, u_n$ be all the nodes in db that are reachable from db's root by some path that matches the regular path expression R, and let $t_1, \ldots, t_n$ denote the labeled graphs with roots $u_1, \ldots, u_n$ respectively: that is, t_i has the same graph as db but with u_i designated as root. Then the meaning of Q is: $t_1 \cup \ldots \cup t_n$. To evaluate Q it suffices to create a new node ρ and add n ε-edges from ρ into $u_1, \ldots, u_n$.

For an example, consider the query:

$$\begin{array}{l}\mathsf{select}\ t \\ \mathsf{where}\ * \Rightarrow CSDept \Rightarrow (not(Dept)) * \Rightarrow Paper \Rightarrow t\ \mathsf{in}\ db\end{array}$$

and the database in Figure 9 (a): the result is in depicted in (b), and in (c) (with ε edges removed). Of course, there is some loss of information, as opposed to keeping the matching nodes separatedly, since we lose track on how edges were grouped according to these nodes. If this grouping is important, we can rephrase the query as:

$$\begin{array}{l}\mathsf{select}\ \{\text{``}Result'' \Rightarrow t\} \\ \mathsf{where}\ * \Rightarrow CSDept \Rightarrow (not(Dept)) * \Rightarrow Paper \Rightarrow t\ \mathsf{in}\ db\end{array}$$

We will discuss in Section 4 such generalizations of regular path expression queries.

3.2 Distributed Evaluation of Regular Queries

We present here a simple distributed evaluation algorithm for regular queries, which is efficient according to Definition 1.1. This is a procedural approach to evaluation on distributed databases, prohibiting further optimizations at each site. In Section 5 we present an algebraic method which allows us to further optimize the query separatedly, at each site.

We start by describing formally a basic evaluation algorithm for regular queries, then extend it to a distributed one. Given a query select t where $R \Rightarrow t$ in db, we denote with A the automaton associated to the regular expression R. Assume A has k states, $States(A) = \{s_1, \ldots, s_k\}$, and that s_1 is it's input state. A's transitions are of the form $s_i \xrightarrow{P} s_j$, where P is a unary predicate on labels or a label constant (see the definition of regular expressions). The algorithm for computing the query on db is shown in Figure 6. At the core is the function *visit* which is a graph traversal function. As it proceeds, it remembers which nodes were visited and which states they were in: we start with the root of db and the initial state s_1. Thus, when we encounter a loop in db, we traverse it at most a number of times equal to the number of states in the automata. In addition we memorize the results of the calls to the function $visit(s, u)$ in a data structure $result[s, u]$. This can be any dictionary data structure, such as binary search tree or hash table. The nodes $u_1, \ldots, u_n$ visited in a terminal state are collected in a set $S = \{u_1, \ldots, u_n\}$. Finally we construct a new node ρ and insert n edges $\rho \xrightarrow{\varepsilon} u_1, \ldots, \rho \xrightarrow{\varepsilon} u_n$.

Next we assume that the database is distributed. First we give a formal definition.

Definition 3.2 *A distributed database is a graph whose nodes are partitioned into m sets, called* sites *or* servers *and denoted s_α, $\alpha = 1, m$. We call a* cross link *an edge $u \to v$ for which u and v are stored on different sites. We assume that all cross links are labeled with ε.*

The assumption that each cross link is an ε edge is not really necessary but it simplifies the presentation. It can be always achieved by replacing every cross link $u \xrightarrow{a} v$ with $u \xrightarrow{a} u' \xrightarrow{\varepsilon} v$, where u' is a fresh node, residing on the same site as u.

Algorithm : *Basic-Evaluation*

Input : A regular path expression R whose automaton is A

A semistructured database db

Output : Evaluates select t where $R \Rightarrow t$ in db

Method : $visited \leftarrow \{\}$

$S \leftarrow visit(InitialState(A), Root(db))$,

Construct the result graph F as follows:

Include all db's nodes and edges in F

Create a new root ρ for F

forall $u \in S$ do

Add a new edge $\rho \xrightarrow{\varepsilon} u$ to F

function $visit(s, u)$

if $(s, u) \in visited$ then return $result[s, u]$

$visited \leftarrow visited \cup \{(s, u)\}$

$result[s, u] \leftarrow \{\}$

if s *is a terminal state* then $result[s, u] \leftarrow \{u\}$

forall $u \xrightarrow{a} v$ (* edge in db *) do

if $a = \varepsilon$ then $result[s, u] \leftarrow result[s, u] \cup visit(s, v)$

else forall $s \xrightarrow{P} s'$ (* automaton transition *) do

if $P(a)$ then $result[s, u] \leftarrow result[s, u] \cup visit(s', v)$

return $result[s, u]$

Figure 6: Basic evaluation algorithm for regular path expressions

Thus, for any given site α, $\alpha = 1, m$, there is a fragment of the graph *db* stored at α, which we denote db_α. We have:

$$\begin{aligned} Nodes(db) &= \bigcup_{\alpha=1,m} Nodes(db_\alpha) \\ Edges(db) &= \bigcup_{\alpha=1,m} Edges(db_\alpha) \cup CrossLinks \end{aligned}$$

For every cross link $u \xrightarrow{\varepsilon} v$ from site α to site β we call u an *output* node in α, and v an *input* node in β. We make the assumption that every site α knows its input and output nodes, $InputNodes(db_\alpha)$, $OutputNodes(db_\alpha)$. This assumption can usually be fulfilled after some preprocessing, depending on how the graph is stored. If it is stored such that for each node we have a list of its outgoing edges (like in the case of web sites), then output nodes are easy to identify, but identifying input nodes requires an additional communication step. By convention, *db*'s root node r is on site 1, $r \in Nodes(db_1)$.

One should notice that a distributed database is different from a collection of views, as in the query answering using views problem, see e.g. [DGMM00]. The views offer a incomplete, and possibly overlapping information about the data. A distributed database is a partition into disjoint fragments, and can be fully reconstructed by taking their union.

Consider now the problem of evaluating a regular query on a distributed database. Executing the algorithm in Figure 6 directly results in the computation being transfered from one site to the other a number of times proportional to the number of cross links. This violates condition 1 of Definition 1.1.

Instead we describe an efficient distributed evaluatin algorithm in Figure 7. The database is distributed on m sites. The idea is simple: each site α traverses only the local graph db_α starting at every input node. There are only two changes from the function *visit* in Algorithm *Basic-Evaluation* to the distributed function $visit_\alpha$. First, when $visit_\alpha$ starts at some input node r of db_α it does not know in which state s that node is reached, if the search were to proceed globally. Hence, to be conservative, $visit_\alpha$ is called on u with all states $s \in States(A)$ (Step 2). Second, when $visit_\alpha$ reaches an output node u of db_α in some state s, it cannot follow its ε link, because it leads to some node u' in a different site, β. Instead α constructs a new output node, which is the pair (s, u). Similarly β constructs a new input node (s, u'), for all its input nodes u' and all states s: the connection between these is done by the client, once the various result fragments F_α, $\alpha = 1, m$, are centralized. Summarizing, each site α constructs a result fragment F_α consisting of: (1) some new input and output nodes of the form (s, u), with u an input, or output node respectively, at site α, and (2) the entire database fragment db_α. The latter cannot be ruled out as being part of the query's result until all fragments $F_1, \ldots, F_m$ are inspected. At this point it is obvious that $Q(db)$ can be obtained (up to bisimulation) by taking the union of $F_1, \ldots, F_m$, adding all missing cross links (i.e. $u \xrightarrow{\varepsilon} u'$ and $(s, u) \xrightarrow{\varepsilon} (s, u')$, with $u \in OutputNodes(db_\alpha), u' \in InputNodes(db_\beta), s \in States(A)$), and defining (s_1, r) to be its root (where r is *db*'s root and s_1 is A's input state). However large parts of the fragments $F_1, \ldots, F_m$ may be inaccessible from the root, and sending them to the client may violate condition 2 of Definition 1.1. For that reason, we do some additional work in Steps 4, 5, 6, in order to compute the accessible part of F_α, for each α. Namely we construct at each site the *accessibility graph*: this has F_α's input and output nodes, and one edge from some input node to some output node if and only if they are connected in F_α. These graphs are sent to the client: note that the total amount of data exchanged is $O(n^2)$, where n is the total number of cross links in *db*. The client assembles these pieces together by adding the missing cross-links, and computes all nodes accessible from (s_1, r). In Step 5 it sends these nodes back to the servers. At this point

Algorithm : *Distributed-Evaluation*

Input : A regular path expression R whose automaton is A

A semistructured database db distributed on a number of sites: $db = \bigcup_\alpha db_\alpha$

Output : Evaluates select t where $R \Rightarrow t$ in db

Method :

Step 1 Send Q to all servers α, $\alpha = 1, m$.

Step 2 At every site α let F_α be db_α:

$\quad Nodes(F_\alpha) \leftarrow Nodes(db_\alpha)$ $InputNodes(F_\alpha) \leftarrow InputNodes(db_\alpha)$

$\quad Edges(F_\alpha) \leftarrow Edges(db_\alpha)$ $OutputNodes(F_\alpha) \leftarrow OutputNodes(db_\alpha)$

$visited_\alpha \leftarrow \{\}$

forall $r \in InputNodes(db_\alpha), s \in States(A)$ do

$\quad S \leftarrow visit_\alpha(s, r)$

$\quad InputNodes(F_\alpha) \leftarrow InputNodes(F_\alpha) \cup \{(s, r)\}$

$\quad$ forall $p \in S$ do $Edges(F_\alpha) \leftarrow Edges(F_\alpha) \cup \{(s, r) \xrightarrow{\varepsilon} p\}$

Step 3 At every site α construct the accessibility graph for F_α (see text)

Step 4 Every site α sends it accessibility graph to the client site.

Compute the global accessibility graph at the client site (see text).

Step 5 Broadcast the global accessibility graph to every server site α, $\alpha = 1, m$.

Step 6 Every site α computes F_α^{acc}, the accessible part of F_α.

Step 7 Every site α sends F_α^{acc} to the client site,

where it is assembled into the result.

function $visit_\alpha(s, u)$

$\quad$ if $(s, u) \in visited_\alpha$ then return $result_\alpha[s, u]$

$\quad visited_\alpha \leftarrow visited_\alpha \cup \{(s, u)\}$

$\quad result_\alpha[s, u] \leftarrow \{\}$

$\quad$ if $u \in OutputNodes(db_\alpha)$

$\qquad$ then $OutputNodes(F_\alpha) \leftarrow OutputNodes(F_\alpha) \cup \{(s, u)\}$

$\qquad\quad result_\alpha[s, u] \leftarrow result_\alpha[s, u] \cup \{(s, u)\}$

$\quad$ else if s is a terminal state then $result_\alpha[s, u] \leftarrow \{u\}$

$\qquad$ forall $u \xrightarrow{a} v$ *in* $Edges(db_\alpha)$ do

$\qquad\quad$ if $a = \varepsilon$ then $result_\alpha[s, u] \leftarrow result_\alpha[s, u] \cup visit_\alpha(s, v)$

$\qquad\quad$ else forall $s \xrightarrow{P} s'$ (* automata transition *) do

$\qquad\qquad$ if $P(a)$ then $result_\alpha[s, u] \leftarrow result_\alpha[s, u] \cup visit_\alpha(s', v)$

$\quad$ return $result_\alpha[s, u]$

Figure 7: Distributed Evaluation Algorithm for Regular Queries

each server α knows its accessible input nodes, so it can compute F_α's accessible part, F_α^{acc}. In Step 7 these parts are gathered at the client: note that here the total size of data sent is $O(r)$, where r is the size of the query's result.

To summarize:

Theorem 3.3 *Algorithm Distributed-Evaluation evaluates efficiently a regular query Q on a distributed database db. Specifically, if n is the number of cross links in db and r is the size of the result $Q(db)$, then:*

1. *The number of communication steps is four (independent on the data or query).*
2. *The total amount of data exchanged during communications has size $O(n^2) + O(r)$.*

Example 3.4 Consider the following regular query:

select t
where $* \Rightarrow$ *"CSDept"* $\Rightarrow$ *not("Dept")* $* \Rightarrow$ *"Paper"* $\Rightarrow t$ in db

The automaton corresponding to this regular query is shown in Figure 8, and has three states, s_1, s_2, s_3. Consider now the database in Figure 9 (a): the query's result is shown in (b) and (c) (with ε edges eliminated). To apply algorithm *Distributed-Evaluation* we start by reorganizing the database as in Figure 10 (a): v_1 is now db's root. Then we compute locally the graphs F_1 and F_2, which are schematically shown in Figure 10 (b) (to reduce clutter we did not show the input nodes $v_1, (s_2, v_1)$, and (s_3, v_1) in F_1, and dropped most of the inner nodes of F_1). We explain some of the edges in F_1: the ε edge from (s_2, v_4') to u_1 is there because there exists a path from v_4' to u_1 in db_1 which matches the transition from s_2 to s_3 in the automaton A, and s_3 is terminal; the ε-edge from (s_2, v_4') to (s_2, v_3) is there because in db_1 there exists a path from v_4' to v_3 matching a transition from s_2 to s_2; the ε-edge from (s_3, v_4') to v_4' is there because s_3 is a terminal state; the edge labeled *Paper* from v_4' to u_1 is simply copied from db_1, etc. Next we compute locally the accessibility graphs: these are just summaries of F_1, F_2, showing which output node is reachable from which input node. They are sent to the client (or any centralized site), which computes all nodes accessible from (s_1, v_1): in Figure 10 (b) these are marked with a surrounding box. Here (s_2, v_2) is first found accessible, which implies (s_2, v_2') accessible too (recall that the client completes the missing ε cross edges). This makes (s_2, v_4) and u_2 accessible. Given the ε edge from v_4 to v_4', we have (s_2, v_4') accessible, hence v_3 and (s_2, v_3) are accessible too; finally (s_2, u_2) is accessible. The accessible input nodes of F_1, F_2 are sent back to the servers, which now start marking the accessible nodes inside F_1 and F_2 respectively. Here u_1 and all its successors will be found accessible in F_1, and all successors of u_2 will be found accessible in F_2. Finally only the accessible fragments of F_1, F_2 are sent to the client, where they are assembled to yield the result. $\square$

4 Select-Where Queries

Most applications require more than just regular path expressions. We discuss here a class of queries which we call select-where queries, in which regular path expressions can be intermixed freely with selections, joins, grouping, and limited data restructuring. This language is a subset of UnQL [BDHS96, BFS00] whose syntax was inspired from OQL [Cat94].

We start with *patterns*, defined by the grammar:

$$P ::= t \mid \{P, \ldots, P\} \mid x \Rightarrow P \mid R \Rightarrow P$$

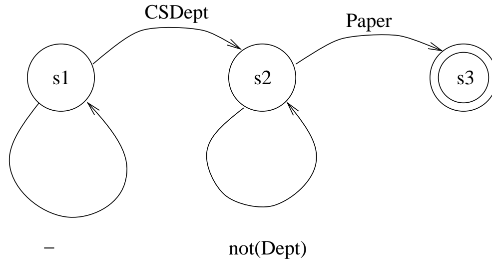


Figure 8: Automaton for Example 3.4

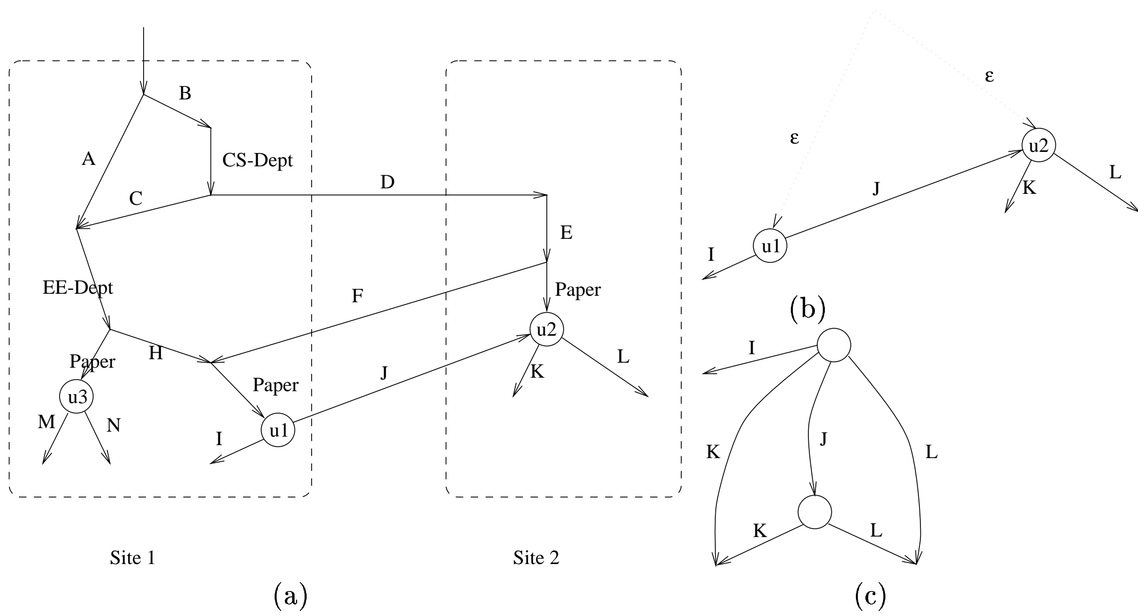
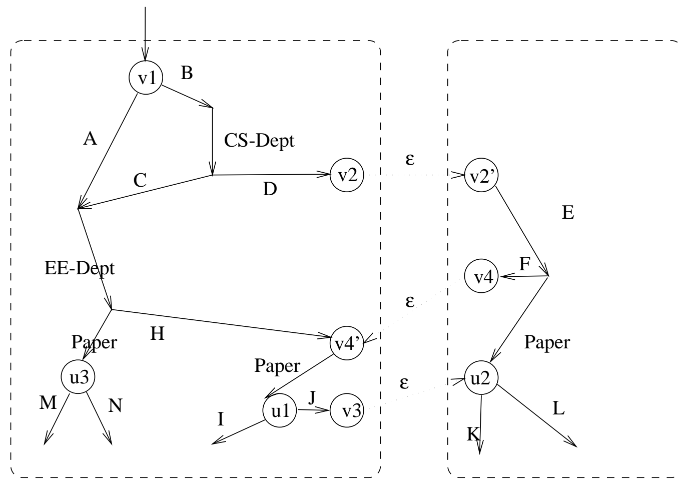


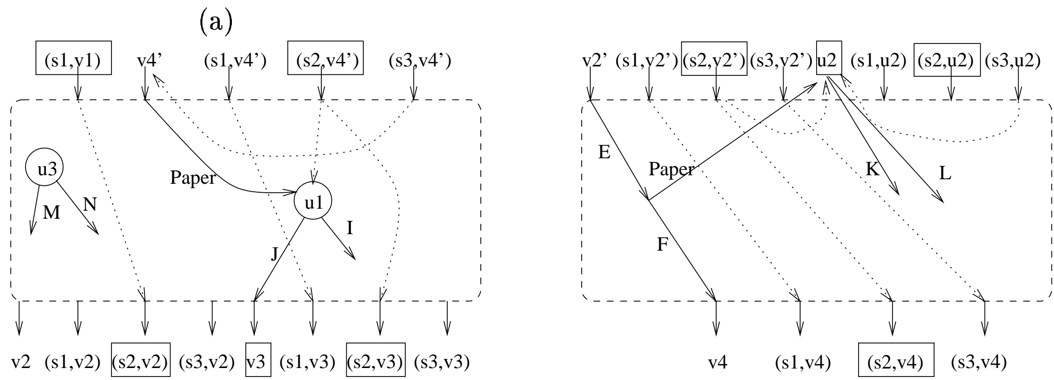
Figure 9: A database and the result of the regular path expression $* \Rightarrow$ *"CSDept"* $\Rightarrow$ *not*(*"Dept"*) $*$ $\Rightarrow$ *"Paper"*.



DB1

DB2

(a)



F1

F2

(b)

Figure 10: Example 3.4 continued.

Here t is a *tree variable*, x is a *label variable*, and R a regular path expression. A **select-where query** is:

$$\begin{array}{l} \mathsf{select}\ E \\ \mathsf{where}\ C_1, \ldots, C_n \end{array}$$

Here $C_1, \ldots, C_n$ are *conditions*, and can be of two kinds. The first kind has the form P in t, with P a pattern and t a tree variable, and is called *generator*. The second is a predicate applied to label variables: this includes equalities between label variables. E is a tree variable t, a tree constructor $\{l_1 \Rightarrow E_1, \ldots, l_k \Rightarrow E_k\}$, a union $E \cup E'$, or another select-where query. The symbol db is a distinguished tree variable denoting the input database.

To see an example, the following query groups together the papers in *CSDept* with their abstracts:

$$\begin{array}{l} \mathsf{select}\ \{``Result" \Rightarrow \{``Paper" \Rightarrow t_1, ``Abstract" \Rightarrow t_2\}\} \\ \mathsf{where}\ * \Rightarrow CSDept \Rightarrow not(Dept) * \Rightarrow Paper \Rightarrow t_1\ \mathsf{in}\ db \\ \qquad * \Rightarrow ``Abstract" \Rightarrow t_2\ \mathsf{in}\ t_1 \end{array}$$

We require all variables occurring in patterns (even in separate patterns of the same query) to be distinct. For the case of edge variables this is no restriction since they can be compared for equality separately. For example instead of select t where $* \Rightarrow x \Rightarrow x \Rightarrow t$ in db we would write select t where $* \Rightarrow x \Rightarrow y \Rightarrow t$ in $db, x = y$. We do not allow however trees to be compared for equality, because in our data model there are no node oid's. Equality of tree variables would rather mean equality of their associated trees, which, in our data model, means bisimulation. This is not a limitation. Note OIDs are just values like the labels sitting on the edges, and can be captured in our model too. For example select t where $``A" \Rightarrow t$ in $db$, $``B" \Rightarrow t$ in db would have to find two edges labeled $``A"$ and $``B"$ respectively leading to two "equal" trees. Testing such an equality (bisimulation) is too expensive for a distributed database[4] and we drop it from our select expressions.

The semantics of a select-where query is the standard, active domain semantics [AHV95]. Given a query $Q =$ select E where $C_1, \ldots, C_n$, define a *valuation* to be a mapping θ sending label variables to labels and tree variables to nodes in db, such that all conditions $C_1, \ldots, C_n$ are satisfied; for that we identify a tree variable t with the rooted graph obtained by redefining the root in db to be t. Let $\theta_1, \ldots, \theta_m$ be all valuations of a query Q. Then its meaning is $E[\theta_1] \cup \ldots \cup E[\theta_m]$.

Select-where queries satisfy the following two identities which we will need in the sequel:

$$\mathsf{select}\ E\ \mathsf{where}\ C_1, \ldots, C_n = \mathsf{select}\ (\mathsf{select}\ E\ \mathsf{where}\ C_{i+1}, \ldots, C_n)\ \mathsf{where}\ C_1, \ldots, C_i \quad (1)$$

$$\mathsf{select}\ (E \cup E')\ \mathsf{where}\ C_1, \ldots, C_n = (\mathsf{select}\ E\ \mathsf{where}\ C_1, \ldots, C_n) \cup (\mathsf{select}\ E'\ \mathsf{where}\ C_1, \ldots, C_n) \quad (2)$$

Select-where queries correspond in a precise sense to the SPJRU algebra on relational databases [AHV95, pp.62], which is the algebra consisting of the operators selection, projection, join, rename, and union. More formally, the following result follows from [BDHS96, BFS00]:

Proposition 4.1 *Let Q be a select-where query, $\mathbf{R}$ be a relational database schema [AHV95, pp.31] and R be a relation schema (i.e. set of attribute names). If for every tree db encoding some relational*

[4] It is PTIME complete [JJM92, GHR95].

database of type $\mathbf{R}$*,* $Q(db)$ *is an encoding of a relation of type* R*, then the restriction of* Q *over inputs encoding databases of type* $\mathbf{R}$ *can be expressed in SPJRU. Conversely, any SPJRU query mapping databases of type* $\mathbf{R}$ *to relations of type* R *can be expressed as a select-where query.*

If we substitute our semistructured data model with XML, the select-where queries correspond to fragments of the main two XML query languages proposed in the literature. XML-QL [DFF+98] is an XML query language consisting essentially of select-where queries plus the use of Skolem functions: select-where queries correspond precisely to the XML-QL fragment without Skolem functions. Quilt [CFR00] extends XML-QL (without Skolem functions) by incorporating aggregate functions from SQL and filter operators from XQL [Rob99]: select-where queries correspond to the Quilt fragment without aggregates and without filter operations.

4.1 Difficulties in Distributed Evaluation of Select-Where Queries

It is not obvious how to extend algorithm *Distributed-Evaluation* from regular queries to arbitrary select-where queries. We illustrate here three kinds of problems, in increasing order of difficulty.

Data Restructuring with Grouping Consider the following query:

$$\begin{aligned} Q1 = \textsf{select } & \{x \Rightarrow \textsf{select } t \\ & \qquad\qquad \textsf{where } * \Rightarrow \textit{“Title”} \Rightarrow t \textsf{ in } t_1\} \\ & \textsf{where } * \Rightarrow \textit{“Paper”} \Rightarrow x \Rightarrow t_1 \textsf{ in } db \end{aligned}$$

The query binds x to each label following a *“Paper”* and t_1 to the target node of x. For each such binding it constructs a new edge labeled x, followed by all subtrees in t_1 under a *“Title”* edge. For example if

$$\begin{aligned} db = \{\ & \mathit{Collection} \Rightarrow \{\mathit{Paper} \Rightarrow \{\mathit{file1.ps} \Rightarrow \{\mathit{Title} \Rightarrow \mathit{Optimizations}\}\}\}, \\ & \mathit{Paper} \Rightarrow \{\mathit{file2.ps} \Rightarrow \{\ \mathit{Heading} \Rightarrow \{\mathit{Title} \Rightarrow \{\mathit{Missing}\}\}, \\ & \qquad\qquad\qquad\qquad\qquad \mathit{subTitle} \Rightarrow \{\mathit{Title} \Rightarrow \{\mathit{NoneGiven}\}\}\}\}\} \end{aligned}$$

then the answer to $Q1$ is:

$$\{\mathit{file1.ps} \Rightarrow \{\mathit{Optimizations}\}, \mathit{file2.ps} \Rightarrow \{\mathit{Missing}, \mathit{NoneGiven}\}\}$$

The query's semantics requires the traversal of paths labeled with the regular path expression $* \Rightarrow$ *“Paper”* $\Rightarrow x \Rightarrow * \Rightarrow$ *“Title”*. But in addition to computing this regular expression, the edge label x is returned as part of the answer. In a distributed database x and its corresponding t may be on different sites. In decomposing the query we must ensure that, after shipping all the result fragments to the client, the right x's are paired with the right t's.

Data Restructuring with Ungrouping Consider:

$$\begin{aligned} Q2 = \textsf{select } & \{x \Rightarrow t\} \\ & \textsf{where } * \Rightarrow \textit{“Paper”} \Rightarrow x \Rightarrow * \Rightarrow \textit{“Title”} \Rightarrow t \textsf{ in } db \end{aligned}$$

As in the previous query this query binds x to any label following a *“Paper”* edge, and t to any node further reachable from x after a *“Title”* edge. The difference is in the way the x's and the t's

are grouped. Here, for every binding of the pair (x, t) a new edge labeled x will be created, possibly repeating the same x several times. For the example database db above, the answer is

$$\{file1.ps \Rightarrow \{Optimizations\}, file2.ps \Rightarrow \{Missing\}, file2.ps \Rightarrow \{NoneGiven\}\}$$

In particular if a paper has no title, it's corresponding x will be dropped from the result: this is unlike the previous query. Decomposing this query is harder than the previous one, because the site discovering x has no information on how many times x needs to be replicated. This information will be available only after all fragments of the result are centralized at the client.

Existential Conditions Consider the query:

$$\begin{aligned} Q3 = \;& \mathsf{select}\ t \\ & \mathsf{where}\ * \Rightarrow \text{"}Paper\text{"} \Rightarrow t\ \mathsf{in}\ db \\ & \qquad * \Rightarrow \text{"}Abstract\text{"} \Rightarrow _\ \mathsf{in}\ t \end{aligned}$$

Like a regular expression query this query returns all trees t reachable by a path labeled $* \Rightarrow$ *"Paper"*; but in addition it filters out some trees, keeping only those who have some path $* \Rightarrow$ *"Paper"*. The decision whether to include t in the result or not cannot be taken locally, at the site containing the root of t. On the other hand, we want to avoid sending t over the network if it is not part of the result: thus, apparently, additional communication steps are necessary, before deciding whether to send t or not, in order to comply with condition 2.

Cartesian Product Consider a more complex query:

$$\begin{aligned} Q4 = \;& \mathsf{select}\ \{x \Rightarrow \{y \Rightarrow t_1, z \Rightarrow t_2\}\} \\ & \mathsf{where}\ * \Rightarrow \text{"}Paper\text{"} \Rightarrow x \Rightarrow t\ \mathsf{in}\ db \\ & \qquad * \Rightarrow \text{"}Title\text{"} \Rightarrow y \Rightarrow t_1\ \mathsf{in}\ t \\ & \qquad * \Rightarrow \text{"}Abstract\text{"} \Rightarrow z \Rightarrow t_2\ \mathsf{in}\ t \end{aligned}$$

The query searches for two regular path expressions:

$$* \Rightarrow \text{"}Paper\text{"} \Rightarrow _ \Rightarrow * \Rightarrow \text{"}Title\text{"} \Rightarrow _$$
$$* \Rightarrow \text{"}Paper\text{"} \Rightarrow _ \Rightarrow * \Rightarrow \text{"}Abstract\text{"} \Rightarrow _$$

However these two paths must share a certain common prefix. To complicate matters, we also do some data restructuring. Suppose that a paper has m *"Title"*s and n *"Abstract"*s. Then the corresponding x has to be replicated $m \times n$ times. Intuitively, $Q4$ constructs a cartesian product.

Join Queries Finally we observe that the select-where queries can express traditional join queries on relational databases represented as trees. We illustrate with the join of the two relations $r1, r2$ of Figure 3.

$$\begin{aligned} & \mathsf{select}\ \{\text{"}tup\text{"} \Rightarrow \{\text{"}m\text{"} \Rightarrow \{m \Rightarrow t_1\}, \text{"}n\text{"} \Rightarrow \{n \Rightarrow t_2\}, \text{"}p\text{"} \Rightarrow \{p \Rightarrow t_3\}, \text{"}q\text{"} \Rightarrow \{q \Rightarrow t_4\}\}\} \\ & \mathsf{where}\ \text{"}r1\text{"} \Rightarrow \text{"}tup\text{"} \Rightarrow \{\text{"}m\text{"} \Rightarrow m \Rightarrow t_1, \text{"}n\text{"} \Rightarrow n \Rightarrow t_2\}\ \mathsf{in}\ db, \\ & \qquad \text{"}r2\text{"} \Rightarrow \text{"}tup\text{"} \Rightarrow \{\text{"}m\text{"} \Rightarrow m' \Rightarrow t_2', \text{"}p\text{"} \Rightarrow p \Rightarrow t_3, \text{"}q\text{"} \Rightarrow q \Rightarrow t_4\}\ \mathsf{in}\ db, \\ & \qquad m = m' \end{aligned}$$

There are no efficient algorithms, in the sense of Definition 1.1, for computing joins on two distributed relations r_1, r_2. When r_1 is on one site and r_2 on another, distributed database systems use *semijoins* [KSS97], which violate condition 2 of Definition 1.1

4.2 Restricted Select-Where Queries

As the previous subsection suggested, some select-where queries are easier to evaluate distributively than others. Here we describe a class of queries which, as we show later, are as easy to evaluate as regular queries.

We define these queries inductively on sub-queries. Sub-queries may have one input tree variable t, and, possibly, one input label variable x, hence we write $Q(t)$ or $Q(x, t)$. The top-level query has input db, sub-queries may have different inputs.

Definition 4.2 *A restricted select-where query with tree variable t and possibly label variable x, in notation $Q(t)$ or $Q(x, t)$, is one of:*

1. t.
2. $\{\}$
3. $\{a \Rightarrow Q_1(x, t)\}$ *where a is either x or a label constant*
4. $Q_1(x, t) \cup Q_2(x, t)$
5. select $Q_1(t_1)$ where $R \Rightarrow t_1$ in t
6. select $Q_1(x_1, t_1)$ where $R \Rightarrow x_1 \Rightarrow t_1$ in $t, P(x_1)$, *where $P(x_1)$ is a unary predicate.*

Here Q_1, Q_2 are themselves restricted select-where queries with variables marked accordingly.

The intuition is the following. If a query select E where P in db is a restricted select-where query, then P may introduce only one tree variable t and, possibly, a label variable x occurring right before t. In addition there are restrictions on how x, t are used in E. Namely we may use them freely in constructors, like $\{a_1 \Rightarrow E_1, \ldots, a_k \Rightarrow E_k\}$, but not inside other select-where sub-queries, except that t may be used in immediate sub-queries like select E' where P' in t. The same property holds recursively, for the sub-queries.

All regular queries are restricted select-where queries. Query $Q1$ of Subsection 4.1 is a restricted select-where query, but queries $Q2, Q3, Q4$ are not. To see a more complex restricted select-where query, consider the following:

```
select {x⇒ ( select {y ⇒ t1}
             where * ⇒ B ⇒ y ⇒ t1 in t) ∪
           ( select {z ⇒ t2}
             where * ⇒ C ⇒ z ⇒ t2 in t)}
where * ⇒ A ⇒ x ⇒ t in db
```

Select-where queries with several generators, like select E where $C_1, \ldots, C_n$ may be equivalent to restricted select-where queries, as a consequence of Equation (1). When this is the case, there is limited variable sharing between the conditions $C_1, \ldots, C_n$: every tree variable defined by C_i must be used immediately in C_{i+1}, and only the tree and label variables bound by the last generator C_n may be used in E.

4.3 Distributed Evaluation of Select-Where Queries

In the reminder of this paper we will prove that all join-free select-where queries can be efficiently evaluated on distributed databases. First we show in Section 5 that a class of queries called *structural recursion* can be evaluated efficiently, and that structural recursion queries can express all restricted select-where queries. Then we show in Section 6 how all join-free select-where queries can be evaluated efficiently, using a new algorithm.

5 Distributed Evaluation of Structural Recursion without Nesting

UnQL is based on *structural recursion*, a programming paradigm that allows one to define mutually recursive functions satisfying certain syntactic restrictions. The restrictions guarantee that the recursion always terminates, not only on trees but also on graphs (with cycles). We show here that functions expressed by structural recursion with certain restrictions (without nesting) can be efficiently evaluated on distributed databases. This has two consequences. First, prior work [BFS00] shows that all select-where queries can be translated into structural recursion queries. Examining that translation we notice here that restricted select-where queries are expressed as structural recursion expressions without nesting. This gives us a method for evaluating restricted select-where queries efficiently on distributed databases. Second, efficient evaluation of structural recursion is of independent interest. The same prior work [BFS00] shows that there exists a tight connection between XSLT [Cla99b] and structural recursion. Namely simple XSLT programs can be expressed as (unnested) structural recursion functions. This gives us an efficient evaluation method for simple XSLT programs on distributed databases.

5.1 Structural Recursion

We review here the definition of structural recursion from [BFS00]. We start by illustrating structural recursion in three examples.

Example 5.1 The following function computes the regular expression $* \Rightarrow Dept$ (recall that $Dept$ is a unary predicate on edge labels):

$$\begin{aligned}
\varphi(\{\}) &= \{\} \\
\varphi(\{x \Rightarrow t\}) &= \mathsf{if}\ Dept(x)\ \mathsf{then}\ t \cup \varphi(t)\ \mathsf{else}\ \varphi(t) \\
\varphi(t \cup t') &= \varphi(t) \cup \varphi(t')
\end{aligned}$$

□

Evaluation proceeds as follows. Start with a tree and check one of three cases. If the tree is empty, apply the first line (return $\{\}$). If the tree is a singleton set then apply the second line: this results in a recursive call of φ on a subtree. Finally, if the tree is not a singleton then decompose it arbitrarily into two trees $t \cup t'$ and apply the function recursively on each of them. The function obviously terminates on every tree. It is less clear at this point what happens if the input argument is a graph with cycles: we show below that the function still terminates in this case too. The reader may check that $\varphi(db)$ returns the same answer as the regular expression $* \Rightarrow Detp$ on the database db. Notice that on an edge x that matches $Dept$ we return $t \cup \varphi(t)$. If we replace the second line with:

$$\varphi(\{x \Rightarrow t\}) = \mathsf{if}\ Dept(x)\ \mathsf{then}\ t\ \mathsf{else}\ \varphi(t)$$

then the function would compute the regular expression $not(Dept) * \Rightarrow Dept$ instead.

The second example illustrates the power of structural recursion in restructuring the input data.

Example 5.2 Consider the function:

$$
\begin{array}{ll}
\varphi(\{\}) & = \{\} \\
\varphi(\{x \Rightarrow t\}) & = \textsf{if } Dept(x) \textsf{ then } \{x \Rightarrow \{\text{"}Name\text{"} \Rightarrow t\}\} \textsf{ else } \{x \Rightarrow \varphi(t)\} \\
\varphi(t \cup t') & = \varphi(t) \cup \varphi(t')
\end{array}
$$

□

A non-department edge x will simply be copied in the answer returned (the $\{x \Rightarrow \varphi(t)\}$ expression). A department edge x will determine an additional edge labeled *"Name"* to be inserted after x. For example, given the data:

$$
\begin{array}{l}
db = \{\ \text{"}Colledge\text{"} \Rightarrow \{\text{"}Eng.\text{"} \Rightarrow \{\ \text{"}Dept\text{"} \Rightarrow \{\text{"}CS\text{"}\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{"}Dept\text{"} \Rightarrow \{\text{"}EE\text{"}\}\}\}, \\
\qquad\quad \text{"}Colledge\text{"} \Rightarrow \{\text{"}Arts\text{"} \Rightarrow \{\ \text{"}Dept\text{"} \Rightarrow \{\text{"}English\text{"}\}\}\}\}
\end{array}
$$

the function $\varphi(db)$ returns:

$$
\begin{array}{l}
\{\ \text{"}Colledge\text{"} \Rightarrow \{\text{"}Eng.\text{"} \Rightarrow \{\ \text{"}Dept\text{"} \Rightarrow \{\text{"}Name\text{"} \Rightarrow \{\text{"}CS\text{"}\}\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{"}Dept\text{"} \Rightarrow \{\text{"}Name\text{"} \Rightarrow \{\text{"}EE\text{"}\}\}\}\}, \\
\quad \text{"}Colledge\text{"} \Rightarrow \{\text{"}Arts\text{"} \Rightarrow \{\ \text{"}Dept\text{"} \Rightarrow \{\text{"}Name\text{"} \Rightarrow \{\text{"}English\text{"}\}\}\}\}\}
\end{array}
$$

We assumed here that the predicate $Dept(x)$ is true only when $x =$ *"Dept"*. This example illustrates how structural recursion can transform a tree, regardless of its shape or depth. The restructuring can be quite dramatic: if the entire database db consists only of $Dept$ edges, then $\varphi(db)$ returns a tree with twice as many edges.

Finally, the third example illustrates the use of multiple structural recursion functions.

Example 5.3 Suppose we have a semistructured database of departments (*"CSDept"*, *"EEDept"*, etc.), containing, among other things, publications like *"Paper"*, *"TR"* (Technical Report), etc. We want to name uniformly all publications in the Computer Science Department as *"TR"*: we want to keep all other publications unchanged. We assume that we have a predicate $Pub(x)$ which checks whether the label x denotes a publication. Then the query constructing the new database can be expressed as $Q(db) = \varphi_1(db)$, where φ_1, φ_2 are two mutually recursive functions defined by:

$$
\begin{array}{llll}
\varphi_1(\{\}) & = \{\} & \varphi_2(\{\}) & = \{\} \\
\varphi_1(\{x \Rightarrow t\}) & = \textsf{if } CSDept(x) \textsf{ then } \{x \Rightarrow \varphi_2(t)\} & \varphi_2(\{x \Rightarrow t\}) & = \textsf{if } Pub(x) \textsf{ then } \{\text{"}TR\text{"} \Rightarrow \varphi_2(t)\} \\
 & \quad \textsf{else } \{x \Rightarrow \varphi_1(t)\} & & \quad \textsf{else } \{x \Rightarrow \varphi_2(t)\} \\
\varphi_1(t \cup t') & = \varphi_1(t) \cup \varphi_1(t') & \varphi_2(t \cup t') & = \varphi_2(t) \cup \varphi_2(t')
\end{array}
$$

□

Here we define two mutually recursive functions, φ_1 and φ_2. The main function, φ_1, is called on the root of the tree and, as it proceeds downwards, it may eventually call the function φ_2 on some subtree.

The general definition is given below.

Definition 5.4 *We say that m mutually recursive functions $\varphi_1, \ldots, \varphi_m$ are defined by structural recursion if, for every $i = 1, \ldots, m$, the function φ_i is defined as follows:*

$$
\begin{aligned}
\varphi_i(\{\}) &= \{\} \\
\varphi_i(\{x \Rightarrow t\}) &= E_i(x, \varphi_1(t), \ldots, \varphi_m(t)) \\
\varphi_i(t \cup t') &= \varphi(t) \cup \varphi(t')
\end{aligned}
$$

where each E_i, $i = 1, \ldots, n$, is an expression that only uses conditionals on the label x and constructors on the values $\varphi_1(t), \ldots, \varphi_m(t)$.

Implicit in this definition is the fact that each function φ_i must be defined by three patterns, corresponding to the empty tree, the singleton tree, and the union of two trees. Moreover, for the first and third pattern the right hand side are required to be $\{\}$ and $\varphi_i(t) \cup \varphi_i(t')$ respectively; in the sequel we will sometimes omit them from the function's definition. This restriction has to do with determinism, ensuring that $\varphi_i(t)$ has the same meaning no matter how we decompose t into the union of two trees. The definition also restricts what recursive calls φ_i may do for the singleton tree case, $\{x \Rightarrow t\}$: it may call any of the other functions $\varphi_1, \ldots, \varphi_m$ recursively (including all of them), but only on the subtree t, and the results of the recursive calls must be combined in an expression using only constructors and conditionals. The constructors are those from Sec. 2.2: $\{\}, \{\mathit{Label} \Rightarrow T\}, T \cup T'$, while the conditionals have the form $\mathsf{if}\ p(x)\ \mathsf{then} - \mathsf{else} -$, with p some predicate.

This definition is a restriction of that given in [BFS00] in that it does not allow for *nested* functions: hence, we will call it structural recursion without nesting. In particular joins are not expressible with this restricted form.

Notice that the first example is not correct according to our definition since it uses the free variable t in the second line. We can always rewrite such functions by "copying" t with an auxiliary function γ. Thus the first example becomes:

$$
\begin{aligned}
\varphi(\{\}) &= \{\} & \gamma(\{\}) &= \{\} \\
\varphi(\{x \Rightarrow t\}) &= \mathsf{if}\ \mathit{CSDept}(x)\ \mathsf{then}\ \gamma(t) \cup \varphi(t)\ \mathsf{else}\ \varphi(t) & \gamma(\{x \Rightarrow t\}) &= \{x \Rightarrow \gamma(t)\} \\
\varphi(t \cup t') &= \varphi(t) \cup \varphi(t') & \gamma(t \cup t') &= \gamma(t) \cup \gamma(t')
\end{aligned}
$$

We will use the copy function in the sequel.

We end this subsection with an important remark relating structural recursion and XSLT [Cla99b], a language supported by the W3C for expressing XML to XML transformations. XSLT is not declarative, but has an execution model that consists of a recursive traversal of the XML tree. There is a close connection between XSLT and structural recursion that is described in [BFS00]: namely a certain core fragment of XSLT corresponds precisely to structural recursion. This is important, since it implies that the efficient distributed evaluation technique we describe here for structural recursion also applies to XSLT programs expressed in that core. Of course, since XSLT is Turing complete, it can express more complex XSLT programs that do not correspond to structural recursion, and the techniques described here do not apply to such programs.

5.2 Translating Restricted Select-Where Queries into Structural Recursion

Select-where queries can be translated into structural recursion: this is shown in [BFS00]. Our goal here is to show that restricted select-where queries can be translated into structural recursion without nesting.

First we need to revise the result from [BFS00] showing that regular expression queries can be translated into structural recursion without nesting. More precisely, every query $Q(t) = \mathsf{select}\ t_1\ \mathsf{where}\ R \Rightarrow t_1\ \mathsf{in}\ t$ is equivalent to $Q(t) = \varphi_1(t)$, where $\varphi_1, \ldots, \varphi_m, \gamma$ are $m+1$ mutually recursive functions, with γ being the copy function. The construction is done by building some automaton for R (it can be nondeterministic), then associating one function φ_i to each state in the automaton. The copy function γ is needed in addition for terminal states. We refer the reader to [BFS00] for the general construct, and only illustrate here an example:

Example 5.5 Consider the regular query

$$Q(db) = \mathsf{select}\ t\ \mathsf{where}\ * \Rightarrow\ CSDept \Rightarrow not(Dept) * \Rightarrow Paper \Rightarrow t\ \mathsf{in}\ db$$

Figure 8 shows the automaton for the regular expression. Then $Q(db) = \varphi_1(db)$, where $\varphi_1, \varphi_2, \varphi_3, \gamma$ are four mutually recursive functions corresponding to the three states in the automaton plus the copy function:

$$\begin{aligned}
\varphi_1(\{\}) &= \{\} \\
\varphi_1(\{l \Rightarrow t\}) &= \mathsf{if}\ CSDept(l) \\
&\quad\ \mathsf{then}\ \varphi_1(t) \cup \varphi_2(t) \\
&\quad\ \mathsf{else}\ \varphi_1(t) \\
\varphi_1(t \cup t') &= \varphi_1(t) \cup \varphi_1(t')
\end{aligned}$$

$$\begin{aligned}
\varphi_2(\{\}) &= \{\} \\
\varphi_2(\{l \Rightarrow t\}) &= \mathsf{if}\ Paper(l) \\
&\quad\ \mathsf{then}\ \varphi_2(t) \cup \varphi_3(t) \cup \gamma(t) \\
&\quad\ \mathsf{else\ if}\ not(Dept(l)) \\
&\quad\quad\quad \mathsf{then}\ \varphi_2(t) \\
&\quad\quad\ \mathsf{else}\ \{\} \\
\varphi_2(t \cup t') &= \varphi_2(t) \cup \varphi_2(t')
\end{aligned}$$

$$\begin{aligned}
\varphi_3(\{\}) &= \{\} \\
\varphi_3(\{l \Rightarrow t\}) &= \{\} \\
\varphi_3(t \cup t') &= \varphi_3(t) \cup \varphi_3(t')
\end{aligned}$$

The idea is that each function $\varphi_i(t)$ corresponds to the regular language obtained by taking state i as initial state, less the empty string. Notice that φ_3 returns the empty set, because there are no outgoing edges from state 3 (hence only the empty string is accepted, if state 3 becomes initial state). The role of γ is to copy the tree being returned once the final state is reached. □

Before giving the translation of restricted select-where queries into structural recursion we start by replacing every query of the form:

$$\mathsf{select}\ Q_1(x_1, t_1)\ \mathsf{where}\ R \Rightarrow x_1 \Rightarrow t_1\ \mathsf{in}\ t, P(x_1)$$

(case 6 in Definition 4.2) with:

$$\mathsf{select}\ (\mathsf{select}\ Q_1(x_1, t_1)\ \mathsf{where}\ x_1 \Rightarrow t_1\ \mathsf{in}\ t_2, P(x_1))\ \mathsf{where}\ R \Rightarrow t_2\ \mathsf{in}\ t$$

Thus, every generator in any where clause will have either the form $R \Rightarrow t_1 \in t$ or the form $x_1 \Rightarrow t_1 \in t$ followed by a predicate $P(x_1)$.

Now we can prove by induction on the structure of a restricted select-where query $Q(x, t)$ that there exist a set of m structural recursive functions $\varphi_1, \ldots, \varphi_m$ and an expression $E(x, t_1, \ldots, t_m)$ consisting only of conditionals on x and constructors on the tree variables $t_1, \ldots, t_m$, such that $Q(x, t) = E(x, \varphi_1(t), \ldots, \varphi_m(t))$. We consider each case in Definition 4.2.

1. When $Q(x, t) = t$, then take $Q(x, t) = \gamma(t)$, where γ is the copy function.
2. When $Q(x, t) = \{\}$ then take $Q(x, t) = \{\}$.

3. When $Q = \{a \Rightarrow Q_1(x,t)\}$, apply the translation inductively on $Q_1(x,t)$ to obtain functions $\varphi_1, \ldots, \varphi_m$ and expression E_1 s.t. $Q_1(x,t) = E_1(x, \varphi_1(t), \ldots, \varphi_m(t))$, then define $Q(x,t) = \{a \Rightarrow E_1(x, \varphi_1(t), \ldots, \varphi_m(t))\}$.

4. When $Q = Q_1 \cup Q_2$, apply the construction inductively to Q_1, Q_2, assume that they use differently named recursive functions $\varphi_1, \ldots, \varphi_m$ and $\psi_1, \ldots, \psi_n$ respectively (otherwise rename), finally define $Q(x,t) = E_1(x, \varphi_1(t), \ldots, \varphi_m(t)) \cup E_2(x, \psi_1(t), \ldots, \psi_n(t))$.

5. When $Q(t) = \mathsf{select}\ Q_1(t_1)\ \mathsf{where}\ R \Rightarrow t_1\ \mathsf{in}\ t$ then we apply the construction above for the regular expression R but replace the copy function with the function computing Q_1. More precisely, let $\varphi_1, \ldots, \varphi_m, \gamma$ be the functions that compute the regular expression R (i.e. $\varphi_1(t)$ is equivalent to $\mathsf{select}\ t_1\ \mathsf{where}\ R \Rightarrow t_1\ \mathsf{in}\ t$) and let $Q_1(t_1) = E(\psi_1(t), \ldots, \psi_n(t))$ be the result of the inductive construction for Q_1. Let $\varphi'_1, \ldots, \varphi'_m$ be the result of substituting all recursive calls to $\gamma(t)$ with $E(\psi_1(t), \ldots, \psi_n(t))$. Then $Q(t) = \varphi'_1(t)$.

6. Finally, when $Q(t) = \mathsf{select}\ Q_1(x_1, t_1)\ \mathsf{where}\ x_1 \Rightarrow t_1\ \mathsf{in}\ t, P(x_1)$, then, after applying the inductive step to Q_1, add a new function φ to the mutually recursive functions defined to be:

$$
\begin{aligned}
\varphi(\{\}) &= \{\} \\
\varphi(\{x_1 \Rightarrow t_1\}) &= \mathsf{if}\ P(x_1)\ \mathsf{then}\ Q_1(x_1, t_1)\ \mathsf{else}\ \{\} \\
\varphi(t \cup t') &= \varphi(t) \cup \varphi(t')
\end{aligned}
$$

Example 5.6 Consider the following restricted select-where query:

$$
\begin{aligned}
&\mathsf{select}\ \{x \Rightarrow (\mathsf{select}\ t'\ \mathsf{where}\ \text{“}B''\Rightarrow t'\ \mathsf{in}\ t) \cup t\} \\
&\mathsf{where}\ x \Rightarrow t\ \mathsf{in}\ db
\end{aligned}
$$

This can be expressed as $\varphi_1(db)$, where:

$$
\begin{aligned}
&\varphi_1(\{x \Rightarrow t\})\ = \{x \Rightarrow \varphi_2(t) \cup \gamma(t)\} \\
&\varphi_2(\{x \Rightarrow t'\}) = \mathsf{if}\ x = \text{“}B''\ \mathsf{then}\ \gamma(t')\ \mathsf{else}\ \{\}
\end{aligned}
$$

and γ is the copy function (we omitted the clauses $\varphi_i(\{\}) = \{\}$ and $\varphi_i(t \cup t') = \varphi_i(t) \cup \varphi_i(t')$). $\square$

5.3 Operations on Graphs

It is shown in [BFS00] that structural recursion is equivalent to some algebraic operations over the input tree (or input graph) that do not use recursion. This offers an alternative way to evaluate structural recursion, called *bulk evaluation*, or bulk semantics, that treats the graph as a whole. In particular bulk evaluation applies uniformly to trees and graphs with cycles, without entering infinite loops. The standard, recursive evaluation is called *top-down* evaluation, or top-down semantics[5]. The two evaluations are shown to return the same answer.

The distributed evaluation algorithm for structural recursion is based on the bulk semantics, hence we need to review it here. In this section we start by reviewing the operators on graphs that are needed to express bulk evaluation.

[5]Top-down semantics can also be defined on cyclic data.

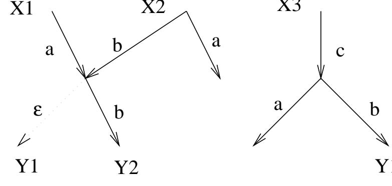


Figure 11: A database with inputs $\mathcal{X} = \{\&x_1, \&x_2, \&x_3\}$ and outputs $\mathcal{Y} = \{\&y_1, \&y_2\}$.

Graphs with Multiple Inputs and Outputs We start by extending our data model to allow certain nodes to be designated input nodes and others output nodes. The names we give to these nodes are called *markers*. Let $Marker$ be an infinite set of markers that we designate $\&x, \&y, \&z, \ldots$. There is a distinguish marker $\& \in Marker$. Let $\mathcal{X}, \mathcal{Y}$ be two finite sets of markers.

Definition 5.7 *A labeled graph with input markers $\mathcal{X}$ and output markers $\mathcal{Y}$ is $g = (V, E, I, O)$ where V is a set of nodes, $E \subseteq V \times (Label \cup \{\varepsilon\}) \times V$ are the edges, $I \subseteq \mathcal{X} \times V$ are the inputs, and $O \subseteq V \times \mathcal{Y}$ are the outputs, such that:*

- *I is an one-to-one mapping from $\mathcal{X}$ to V. For $\&x \in \mathcal{X}$ we denote $v = I(\&x)$ the unique node $v \in V$ s.t. $(\&x, v) \in I$ and call v an input node.*
- *O is a many-to-many mapping from V to $\mathcal{Y}$. Whenever $(v, \&y) \in O$ we call v an output node.*
- *No edges are leaving from output nodes.*

We denote with $DB_{\mathcal{Y}}^{\mathcal{X}}$ the set of labeled graphs with input markers $\mathcal{X}$ and output markers $\mathcal{Y}$. When $\mathcal{X} = \{\&\}$, then we abbreviate $DB_{\mathcal{Y}}^{\mathcal{X}}$ with $DB_{\mathcal{Y}}$. The set $DB_{\emptyset}$ is further abbreviated with DB.

Figure 11 contains an example of a graph with inputs $\&x_1, \&x_2, \&x_3$ and outputs $\&y_1, \&y_2$, which is written as[6] $(\&x_1 := \{a \Rightarrow (\&y_1 \cup \{b \Rightarrow \&y_2\})\}; \&x_2 := \{b \Rightarrow \{\&y_1 \cup \{b \Rightarrow \&y_2\}\}, a\}; \&x_3 := \{c \Rightarrow \{a, b \Rightarrow \&y_1\}\})$.

Graphs with input and output markers generalize the rooted labeled graphs defined in Sec. 2. A rooted graph should be thought as having a single input marker, $\mathcal{X} = \{\&\}$, and no output markers, $\mathcal{Y} = \emptyset$. Hence the notation DB used there is consistent with that used here. Notice that if $g \in DB_{\mathcal{Y}}^{\mathcal{X}}$ and $\mathcal{Y} \subseteq \mathcal{Y}'$ then $g \in DB_{\mathcal{Y}'}^{\mathcal{X}}$. By contrast, when $\mathcal{X} \neq \mathcal{X}'$ then $DB_{\mathcal{Y}}^{\mathcal{X}} \cap DB_{\mathcal{Y}}^{\mathcal{X}'} = \emptyset$. Equality is defined in terms of bisimulation, as before. The definition of a bisimulation (Def. 2.1) extends to two graphs $g, g' \in DB_{\mathcal{Y}}^{\mathcal{X}}$ by replacing condition (1), which required that the two roots be in the bisimulation, with:

(1') For every $\&x \in \mathcal{X}$, $(I(\&x), I'(\&x)) \in R$.

It follows that a graph g is equal to it's accessible part, defined as the collection of edges and nodes accessible from some input node. When $\mathcal{X} = \emptyset$ then there are no input nodes, and any graph in $DB_{\mathcal{Y}}^{\emptyset}$ is equivalent to $(\emptyset, \emptyset, \emptyset, \emptyset)$: this is the empty graph and we denote it with $()$.

Graph Constructors The graph operators we consider consists of nine graph constructors plus structural recursion. We define the graph constructors here. They are:

[6] To be precise, this graph is *bisimilar* to that in Figure 11.

$\{\}$	empty tree
$\{l : d\}$	singleton tree
$d_1 \cup d_2$	union of two trees
$\&x := d$	label the root node with some input marker
$\&y$	data graph with one output marker
$()$	empty data graph
$d_1 \oplus d_2$	disjoint union
$d_1 @ d_2$	append of two data graphs
$\mathsf{cycle}\ (d)$	data graph with cycles

We explain the constructors here, their formal definition is in Fig. 12. Each operator is "typed", in the sense that it expects certain input/output markers and returns certain input/output markers. Recall our convention by which a tree has the "default" input marker &.

The first three are the tree operators: $\{\}$, $\{l : d\}$, $d_1 \cup d_2$. All three constructors expect $d, d_1, d_2 \in DB_{\mathcal{Y}}$ and return results in $DB_{\mathcal{Y}}$.

The next four constructors allow us to create and add input and output markers: $\&x := d$ takes $d \in DB_{\mathcal{Y}}$ and relabels the root with the input marker $\&x$, hence the result is in $DB_{\mathcal{Y}}^{\{\&x\}}$. The constructor $\&y$ returns a tree with a single node labeled with the output marker $\&y$ and default input marker &: hence $\&y \in DB_{\mathcal{Y}}$, where $\&y \in \mathcal{Y}$. This is like $\{\}$, but now we have an output marker on the unique node. The empty graph is denoted by $()$: it has no nodes, no edges, and is the unique graph in $DB_{\mathcal{Y}}^{\emptyset}$. Notice the distinction between the empty graph $()$ and the empty tree $\{\}$ which contains a single node (the root). The disjoint union $d_1 \oplus d_2$ requires d_1 and d_2 to have disjoint sets of input markers $\mathcal{X}_1, \mathcal{X}_2$, and the same set of output markers $\mathcal{Y}$; then $d_1 \oplus d_2 \in DB_{\mathcal{Y}}^{\mathcal{X}_1 \cup \mathcal{X}_2}$.

Finally, the last two constructors deal with the vertical structure of the data graphs. The append operator $d_1 @ d_2$ is defined when $d_1 \in DB_{\mathcal{Y}}^{\mathcal{X}}$ and $d_2 \in DB_{\mathcal{Z}}^{\mathcal{Y}}$, and results in a graph in $DB_{\mathcal{Z}}^{\mathcal{X}}$. Append is essentially performed by gluing each output node in d_1 with the input node in d_2 labeled with the same marker: formally, however, this is achieved by adding ε edges, see Fig. 12. It corresponds to list concatenation, if linear trees are identified with lists: when $d_1 = \{a_1 : \{a_2 : \ldots \{a_n : \&z\} \ldots\}\}$ and $d_2 = (\&z := \{b_1 : \{b_2 : \ldots\}\})$, then $d_1 @ d_2 = \{a_1 : \{a_2 : \ldots \{a_n : \{b_1 : \{b_2 : \ldots\}\}\} \ldots\}\}$. For another illustration, consider $d_1 = \{a : \&y_1,\ b,\ c : \&y_2\}$ and $d_2 = (\&y1 := \{d\},\ \&y2 := \{e, f\})$. Then $d_1 @ d_2$ is value equivalent[7] to $\{a : \{d\},\ b,\ c : \{e, f\}\}$. The effect is that of simultaneously substituting each output marker in d_1 with the value of the corresponding input marker in d_2. The last operator allows us to introduce cycles: when $d \in DB_{\mathcal{X} \cup \mathcal{Y}}^{\mathcal{X}}$, then $\mathsf{cycle}\ (d) \in DB_{\mathcal{Y}}^{\mathcal{X}}$.

We will use the following syntactic sugar. We abbreviate $(\&x_1 := d_1) \oplus \ldots \oplus (\&x_m := d_m)$ with $(\&x_1 := d_1; \ldots; \&x_m := d_m)$. For $\mathcal{X} = \{\&x_1, \ldots, \&x_m\}$, we let $\{\}$ be an abbreviation for "the empty tree" in $DB_{\mathcal{Y}}^{\mathcal{X}}$, defined as $(\&x_1 := \{\}; \ldots; \&x_m := \{\})$. For $d, d' \in DB_{\mathcal{Y}}^{\mathcal{X}}$, $d = (\&x_1 := d_1; \ldots; \&x_m := d_m)$, $d' = (\&x_1 := d'_1; \ldots; \&x_m := d'_m)$ we define $d \cup d'$ to be $(\&x_1 := d_1 \cup d'_1; \ldots; \&x_m := d_m \cup d'_m)$.

As an example of how the graph constructors can be used we show how to express formally the distribution of a database on several sites. Consider a rooted database db which is stored at m different sites. In Subsection 3.2 we denoted with db_α, $\alpha = 1, m$ the m fragments. Recall that we have identified input and output nodes in each db_α, $\alpha = 1, m$. Then db can be represented as:

$$db \quad = \quad \&x_1 @ \mathsf{cycle}_{\mathcal{X}}\ (db_1 \oplus \ldots \oplus db_m)$$

[7]When we apply the definition in Fig. 12, $d_1 @ d_2$ results in $\{a : \{\varepsilon : \{d\}\},\ b,\ c : \{\varepsilon : \{e,\ f\}\}\}$.

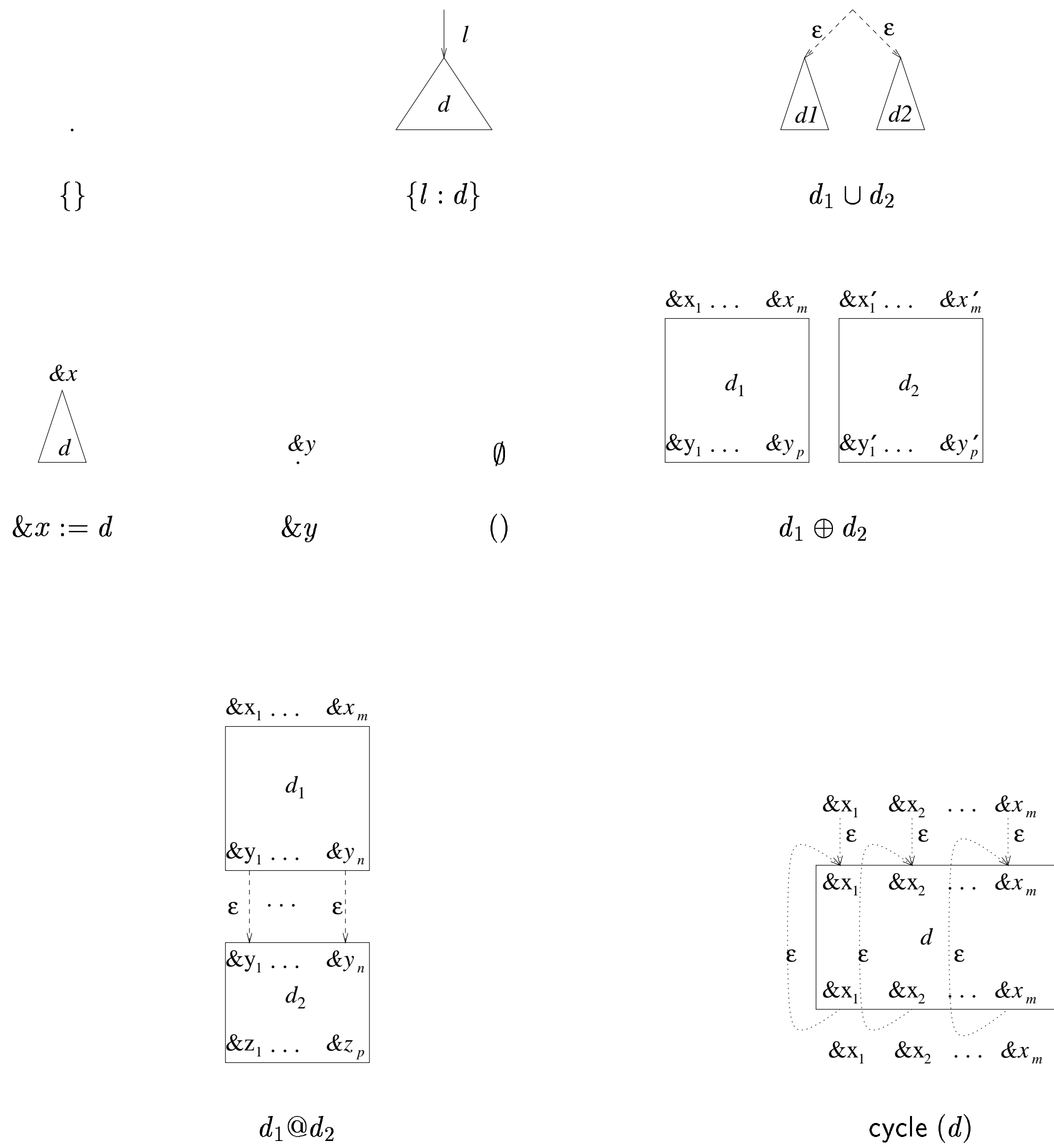


Figure 12: Definition of the constructors.

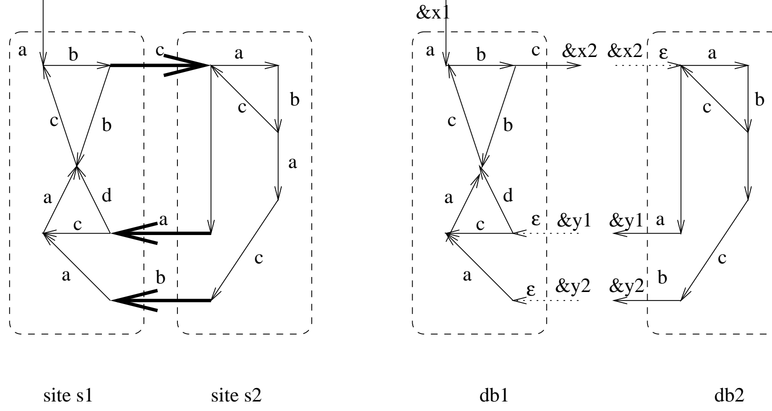


Figure 13: Representation of a distributed database.

Here $\mathcal{X}$ is a set of markers $\mathcal{X} = \{\&x_1, \&x_2, \ldots, \&x_p\}$ whose size is one plus the number of cross links. For example the database *db* in Figure 13 is stored on two sites $\alpha = 1, 2$. We start by cutting the cross links, and inserting markers in place: $\&x_1, \&x_2, \&y_1, \&y_2$, where $\&x_1$ is the input to the old root. Call db_1, db_2 the resulting two fragments: here $db_1 \in DB_{\mathcal{X}}^{\mathcal{X}_1}$ and $db_2 \in DB_{\mathcal{X}}^{\mathcal{X}_2}$, with $\mathcal{X}_1 \stackrel{\text{def}}{=} \{\&x_1, \&y_1, \&y_2\}, \mathcal{X}_2 \stackrel{\text{def}}{=} \{\&x_2\}$, and $\mathcal{X} \stackrel{\text{def}}{=} \mathcal{X}_1 \cup \mathcal{X}_2 = \{\&x_1, \&x_2, \&y_1, \&y_2\}$. $\&x_1$ denotes the root of *db*. Then $db \in DB$ is represented as:

$$db = \&x_1 @ (\mathsf{cycle}_{\mathcal{X}}\ (db_1 \oplus db_2))$$

where $db_1 \oplus db_2$ is the concatenation of db_1 and db_2. The $\mathsf{cycle}_{\mathcal{X}} \ldots$ construct redraws the cross links, while $\&x_1 @ (\ldots)$ selects only the input labeled $\&x_1$ as the unique input of the database.

Structural Recursion in Algebraic Notation The first step towards bulk evaluation is to replace m mutually recursive functions with a single recursive function with m input markers. Formally, let $\varphi_1, \ldots, \varphi_m$ be m mutually recursive functions defined by[8]:

$$\begin{aligned} \varphi_1(\{x \Rightarrow t\}) &= E_1(x, \varphi_1(t), \ldots, \varphi_m(t)) \\ &\ldots \\ \varphi_m(\{x \Rightarrow t\}) &= E_m(x, \varphi_1(t), \ldots, \varphi_m(t)) \end{aligned}$$

Here $E_1(x, t_1, \ldots, t_m), \ldots, E_m(x, t_1, \ldots, t_m)$ are expressions that use only conditionals on x and constructors on the variables $t_1, \ldots, t_m$.

Let $\mathcal{S} = \{\&s_1, \ldots, \&s_m\}$ be a set of m markers and define:

$$E(x) = (\&s_1 := (E_1(x, \&s_1, \ldots, \&s_m); \ldots; \&s_m := E_m(x, \&s_1, \ldots, \&s_m)))$$

E is a function $Label \rightarrow DB_{\mathcal{S}}^{\mathcal{S}}$ that, when given a label x, returns a graph with input markers $\mathcal{S}$ and output markers $\mathcal{S}$. Define the following recursive function φ:

$$\begin{aligned} \varphi(\{\}) &= \{\} \\ \varphi(\{x \Rightarrow t\}) &= E(x) @ \varphi(t) \\ \varphi(t \cup t') &= \varphi(t) \cup \varphi(t') \end{aligned} \tag{3}$$

[8]We omit the definitions $\varphi_i(\{\}) = \{\}$ and $\varphi_i(t \cup t') = \varphi_i(t) \cup \varphi_i(t')$ for $i = 1, m$.

When evaluated recursively on an input tree $t \in DB$, $\varphi(t)$ returns a graph in $DB_\emptyset^{\mathcal{S}}$. Recall our convention that we use in the first and third line: $\{\}$ actually denotes $(\&s_1 := \{\}; \ldots; \&s_m := \{\})$, while $t \cup t'$ is done component wise. The following identity holds for every tree t:

$$(\&s_1 := \varphi_1(t); \ldots; \&s_m := \varphi_m(t)) \quad = \quad \varphi(t)$$

This can be checked easily by induction on t. In particular, if we only want to compute $\varphi_1(t)$, we can obtain it from φ by:

$$\varphi_1(t) \quad = \quad \&s_1 @ \varphi(t)$$

Hence we will assume in the rest of this section that a structural recursion defines a single recursive function, as in Eq.(3). Moreover, we shall denote such a function as:

$$\varphi(t) \quad = \quad rec_{\mathcal{S}}(E)(t) \tag{4}$$

To illustrate, consider the two recursive functions in Example 5.3. Following the construction above we define:

$$\begin{aligned} E(x) \stackrel{\text{def}}{=} (\ &\&s_1 := \mathsf{if}\ l = \text{“}CSDept\text{”}\ \mathsf{then}\ \{l \Rightarrow \&s_2\}\ \mathsf{else}\ \{l \Rightarrow \&s_1\}; \\ &\&s_2 := \mathsf{if}\ Pub(l)\ \mathsf{then}\ \{\text{“}TR\text{”} \Rightarrow \&s_2\}\ \mathsf{else}\ \{l \Rightarrow \&s_2\}) \end{aligned}$$

Then the query $Q(db)$ in Example 5.3 (the function φ_1) can be expressed as:

$$\&s_1 @ rec_{\{\&s_1, \&s_2\}}(E)(db)$$

Bulk Evaluation of Structural Recursion Given an input tree t, the bulk evaluation of the function $\varphi(t) = rec_{\mathcal{S}}(t)$ proceeds as follows. First, each node u in the input tree t will be replaced with m copies, $u_1, \ldots, u_m$ (recall that m is the number of markers in $\mathcal{S}$). Next the function $E(x)$ is applied to each edge $u \xrightarrow{a} v$ in the input tree t, resulting in a graph $E(a)$, with m input nodes and m output nodes: then ε edges are added from $u_1, \ldots, u_m$ to the input nodes of $E(a)$, and ε edges are added from the output nodes of $E(a)$ to $v_1, \ldots, v_m$. Finally, if r was the root in t, then $r_1, \ldots, r_m$ will be designated as the input nodes and labeled with the markers $\&s_1, \ldots, \&s_m$ respectively. For a more detailed discussion we refer the reader to [BFS00]. Notice that the bulk semantics is oblivious to whether the graph has cycles or not, hence it gives meaning to structural recursion on cyclic data.

For an illustration, consider the query Q in Example 5.3, given by two mutually recursive functions φ_1, φ_2. It can be written as:

$$Q(db) = \&s_1 @ rec(E)(db)$$

where:

$$\begin{aligned} E(x) \stackrel{\text{def}}{=} (\ &\&s_1 := \mathsf{if}\ CSDept(x)\ \mathsf{then}\ \{x \Rightarrow \&s_2\}\ \mathsf{else}\ \{x \Rightarrow \&s_1\}; \\ &\&s_2 := \mathsf{if}\ Pub(x)\ \mathsf{then}\ \{\text{“}TR\text{”} \Rightarrow \&s_2\}\ \mathsf{else}\ \{x \Rightarrow \&s_2\}) \end{aligned}$$

Consider now the database db in Figure 14 (a), the bulk evaluation of $rec_{\mathcal{S}}(E)$ is shown in (b). The simplified form (under bisimulation) of $\&s_1 @ rec_{\mathcal{S}}(E)(db)$ is shown in (c).

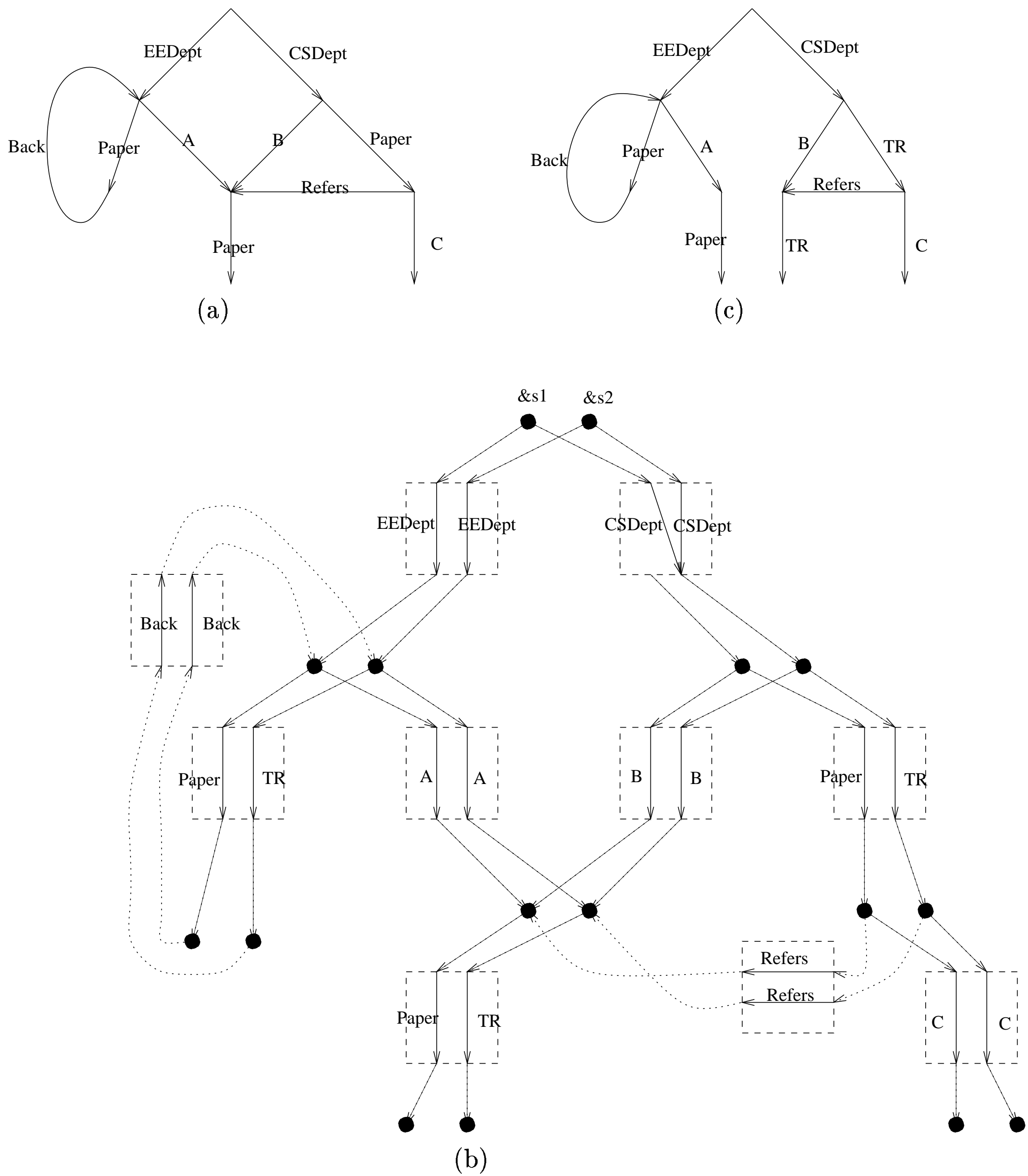


Figure 14: Illustration for Example 5.5. A Database (a), the result of $rec_{\mathcal{S}}(E)(db)$ (b), and the simplified answer of $Q(db) = \&s_1 @\, rec(E)(db)$ (c).

Structural Recursion on Graphs with Input and Output Markers Consider again the structural recursion definition in (4). We have defined its meaning for the case when $t \in DB$. One can extend the definition in a natural way to the case when $t \in DB_{\mathcal{Y}}^{\mathcal{X}}$. For that we need a *marker constructor*: if $\&u$ and $\&s$ are markers, then $\&u \cdot \&s$ is a fresh new marker. One can think of "$\cdot$" as a Skolem function, taking two input oids and returning a fresh output oid. Further, we assume that $\&$ is both a left and right identity: $\& \cdot \&s = \&s \cdot \& = \&s$. Given two sets of markers $\mathcal{U}, \mathcal{S}$, we define $\mathcal{U} \cdot \mathcal{S} = \{\&u \cdot \&s \mid \&u \in \mathcal{U}, \&s \in \mathcal{S}\}$.

Consider now some $t \in DB_{\mathcal{Y}}^{\mathcal{X}}$, where $\mathcal{X} = \{\&x_1, \ldots, \&x_n\}$, $\mathcal{Y} = \{\&y_1, \ldots, \&y_k\}$. We define the result of $rec_{\mathcal{S}}(E)(t)$ to be a graph in $DB_{\mathcal{Y} \cdot \mathcal{S}}^{\mathcal{X} \cdot \mathcal{S}}$ constructed as follows. First apply the bulk construction as in the case of a graph in DB: for each node u in t we have m copies, $u_1, \ldots, u_m$ in the output graph, and there are additional nodes and edges obtained by applying E independently on each edge. Then, for each input node u in t that is labeled with some input marker $\&x_i$, $i = 1, \ldots, n$, label the corresponding nodes $u_1, \ldots, u_n$ with the input markers $\&x_i \cdot \&s_1, \ldots, \&x_i \cdot \&s_m$ respectively. Similarly, for each output node u in t that labeled with some output marker $\&y_i$, $i = 1, \ldots, k$, label the corresponding nodes $u_1, \ldots, u_n$ with the output markers $\&y_i \cdot \&s_1, \ldots, \&y_i \cdot \&s_m$ respectively.

Thus structural recursion treats each input separately from the others. This can be captured precisely by the equation:

$$rec_{\mathcal{S}}(E)(t \oplus t') \quad = \quad rec_{\mathcal{S}}(E)(t) \oplus rec_{\mathcal{S}}(E)(t') \tag{5}$$

Interestingly, the recursive semantics can also be easily adapted to handle output markers in the same way as the bulk semantics. Consider some graph t with output markers $\&y_1, \ldots, \&y_k$, and assume m mutually recursive functions $\varphi_1(t), \ldots, \varphi_m(t)$ to be defined by structural recursion, as in Def. 5.4. To deal with the output markers it suffices to extend their definitions to:

$$\begin{aligned}
\varphi_i(\&y_j) &= \&y_j \cdot \&s_i \\
\varphi_i(\{\}) &= \{\} \\
\varphi_i(\{x \Rightarrow t\}) &= E_i(x, \varphi_1(t), \ldots, \varphi_m(t)) \\
\varphi_i(t \cup t') &= \varphi(t) \cup \varphi(t')
\end{aligned}$$

5.4 A Calculus for Graphs

The calculus $\mathcal{C}$ we consider for computing graphs consists of all constructors in Sec. 5.3, the structural recursion operator $rec_{\mathcal{S}}(E)(t)$, and the conditional $\textsf{if } P(x) \textsf{ then } t \textsf{ else } t'$, where $P(x)$ is a unary predicate on x.

The following is a consequence of our discussion in Sec. 5.2.

Proposition 5.8 *Every restricted select-where query $Q(db)$ can be expressed in $\mathcal{C}$ as $P @ rec_{\mathcal{S}}(E)(db)$, for some set of markers $\mathcal{S}$, where P is a constant expression.*

As a trivial illustration, let us continue Example 5.6. The restricted select-where query Q can be expressed as $Q(db) = \&s_1 @ rec_{\mathcal{S}}(E)(db)$, where:

$$\begin{aligned}
E(x) = (\ &\&s_1 := \{x \Rightarrow \&s_2 \cup \&s_3\}; \\
&\&s_2 := \textsf{if } x = \text{“}B\text{''} \textsf{ then } \&s_3 \textsf{ else } \{\}; \\
&\&s_3 := \{x \Rightarrow \&s_3\})
\end{aligned}$$

5.5 Decomposed Queries and Distributed Evaluation

The idea behind efficient evaluation of queries in $\mathcal{C}$ on distributed databases is to transform them into *decomposed* queries.

Definition 5.9 *Let Q be a query in $\mathcal{C}$. We say that Q is* **decomposed** *iff:*

1. *For all t, t', $Q(t@t') = Q(t)@Q(t')$, and*
2. *For all t, t', $Q(t \oplus t') = (Q(t) \oplus Q(t'))$.*

If $Q(t) = rec_{\mathcal{S}}(E)(t)$, then Q is decomposed: condition 1 follows from [BFS00], while condition 2 is just Equation (5). Some queries in $\mathcal{C}$ fail to be decomposed because the equations in Def. 5.9 do not typecheck (they have the wrong markers). For example consider the query $Q(t) = \{``A'' \Rightarrow t\}$ that adds an *"A"*-edge on top of the database t. Here t may have output markers, but its input marker must be & (by definition of the operator $\{- \Rightarrow -\}$). Equation 2 fails simply because both $Q(t)$ and $Q(t')$ have the same input marker, &, hence the operation $\oplus$ is not technically valid. Equation (1) also fails. First, let's establish the types. For $t@t'$ to make sense we must have $t \in DB_{\mathcal{Y}}$ and $t' \in DB^{\mathcal{Y}}_{\mathcal{Z}}$; for $Q(t')$ to make sense we must have $\mathcal{Y} = \{\&\}$. Assume & occurs exactly once as an output marker in t. Then $t@t'$ is t with that marker substituted with t': $Q(t@t')$ adds a single $``A''$ edge to the top of $t@t'$, while $Q(t)@Q(t')$ has two $``A''$ edges, one at the top, the other above t' root. The two are not equal.

We first show the connection between decomposed queries and efficient computation on distributed databases.

Proposition 5.10 *If Q is decomposed then:*

$$Q(\mathsf{cycle}\ (t)) \quad = \quad \mathsf{cycle}\ (Q(t))$$

Proof: We only give an informal argument. Since $\mathsf{cycle}\ t$ is bisimilar to the infinite unfolding $t@t@\ldots$, we have $Q(\mathsf{cycle}\ t) = Q(t@t@\ldots) = Q(t)@Q(t)@\ldots = \mathsf{cycle}\ Q(t)$. □

This suggests the following evaluation method. Recall that a distributed database can be expressed as:

$$db \quad = \quad \&x_1@\mathsf{cycle}_{\mathcal{X}}\ (db_1 \oplus \ldots \oplus db_m)$$

where $db_1, \ldots, db_m$ are the fragments on the m sites. Then, for a decomposed query Q, we have:

$$Q(db) \quad = \quad Q(\&x_1)@\mathsf{cycle}_{\mathcal{X}}\ (Q(db_1) \oplus \ldots \oplus Q(db_m))$$

We will describe a distributed algorithm based on this equation.

Before that we must address the problem that not every query in $\mathcal{C}$ is decomposed. We show however that each query in $\mathcal{C}$ can be rewritten into a decomposed query followed by some residual computation consisting only of constructors. We give this proposition first, then show how such queries can be efficiently evaluated.

Proposition 5.11 *For every query Q in $\mathcal{C}$ there exists a decomposed query Q^{dec} and constant expression P such that $Q(db) = P@Q^{dec}(db)$ for every graph $db \in DB$.*

We call the process of finding P, Q^{dec} for a given Q "query decomposition". Two important classes of queries we have considered earlier are already in the form given by the proposition and do not need to be further decomposed. The first class are that of restricted select-where queries: recall from Prop. 5.8 that every restricted select-where query can be rewritten as $P@\,rec_{\mathcal{S}}(E)(db)$, which is in the form given in the proposition. The second class are those given by structural recursion, like $Q(db) = \varphi_1(db)$, where $\varphi_1(t), \ldots, \varphi_m(t)$ are m mutually recursive functions defined by structural recursion. This includes queries expressed in a certain core fragment of XSLT (see the discussion at the end of Sec. 5.1). We have seen that Q can be rewritten as $Q(db) = \&x_1@\,rec(E)(db)$: this is also in the form given by the proposition, and does not need to be further decomposed.

Proof: (of Proposition 5.11) We prove by induction on the structure of Q in $\mathcal{C}$. Queries in $\mathcal{C}$ have a relatively simple structure, since structural recursions cannot be nested. That is, in $rec_{\mathcal{S}}(E)(t)$, the expression E has only constructors and conditionals, and no structural recursion; hence, we do not need to extend the induction to E. This implies that we do not need to treat conditionals ($\texttt{if}$ $P(x)$ $\texttt{then}$ Q $\texttt{else}$ Q'), since they can only occur inside some $rec_{\mathcal{S}}(E)$ expression. Another simplifying assumption is in the treatement of $Q_1(db)@Q_2(db)$. here we use the fact that db has no output markers (since $db \in DB$). As a consequence $Q_1(db)$ cannot use db in any interesting way. For example, the expression $\{a{\Rightarrow}(db{\cup}\{b{\Rightarrow}db\})\}@Q_2(db)$ is equal to $\{a{\Rightarrow}(db{\cup}\{b{\Rightarrow}db\})\}$ since there is no output marker to append with $Q_2(db)$. For another example, consider $\{a{\Rightarrow}(db{\cup}\{b{\Rightarrow}\&y\})\}@Q_2(db)$: this expression is equal to $\{a \Rightarrow (db \cup \{b \Rightarrow (\&y@Q_2(db))\})\}$ and, again, db in the first query does not contribute to the append operation. Hence we will restrict our induction to the case when $Q_1(db)$, in $Q_1(db)@Q_2(db)$, is a constant query (i.e. does not depend on db). Finally, we pre-process the entire query Q such as to rename the local markers $\mathcal{S}$ in every $rec_{\mathcal{S}}$ construct to be disjoint from all other markers used in other rec constructs. With these observations in mind, we proceed to the proof by induction.

1. $Q(db) = db$. Then take $P \stackrel{\text{def}}{=} (\& := \&)$ and $Q^{dec}(t) \stackrel{\text{def}}{=} t$. We obviously have $Q(db) = P@Q^{dec}(db)$.

2. $Q(db) = \{\}$. Then take $P \stackrel{\text{def}}{=} \{\}$ and $Q^{dec} \stackrel{\text{def}}{=} ()$.

3. $Q(db) = \{a \Rightarrow Q_1(db)\}$. Here a can only be a label constant, not a label variable. First apply induction hypothesis to Q_1 to decompose it into $Q_1(db) = P_1@Q_1^{dec}(db)$. Then $P \stackrel{\text{def}}{=} \{a \Rightarrow P_1\}$ and $Q^{dec} \stackrel{\text{def}}{=} Q_1^{dec}$.

4. $Q(db) = Q_1(db) \cup Q_2(db)$. First apply induction hypothesis to decompose $Q_1(db) = P_1@Q_1^{dec}(db)$ and $Q_2(db) = P_2@Q_2^{dec}(db)$. The idea is to define $P \stackrel{\text{def}}{=} P_1 \cup P_2$ and $Q^{dec}(db) \stackrel{\text{def}}{=} (Q_1^{dec}(db) \oplus Q_2^{dec}(db))$. But we must take some precautions to make sure that $(Q_1^{dec}(db) \oplus Q_2^{dec}(db))$ is well defined, i.e. that the two queries have distinct input markers. A careful analysis of the induction process (using the fact that all rec constructs have disjoint sets of markers) shows that $Q_1^{dec}(db)$ can either (a1) only have private input markers, not shared with $Q_2^{dec}(db)$, or (b1) be of the form $((Q_1^{dec})'(db) \oplus db)$. Similarly, $Q_2^{dec}(db)$ can either (a2) have only private input markers, or (b2) be of the form $((Q_2^{dec})'(db) \oplus db)$. For the first three of the four combined cases, it is safe to define $Q^{dec}(db) \stackrel{\text{def}}{=} (Q_1^{dec}(db) \oplus Q_2^{dec}(db))$. For the last case we define $Q^{dec}(db) \stackrel{\text{def}}{=} ((Q_1^{dec})'(db) \oplus (Q_2^{dec})'(db) \oplus db)$.

5. $Q(db) = Q_1(db)@Q_2(db)$. Based on our remark, it suffices to consider only the case when Q_1 is a constant expression. We apply induction hypothesis first to Q_2, $Q_2(db) = P_2@Q_2^{dec}(db)$, then define $P \stackrel{\text{def}}{=} Q_1@P_2$, $Q^{dec}(db) \stackrel{\text{def}}{=} Q_2^{dec}(db)$.

6. $Q(db) = rec_{\mathcal{S}}(E)(Q_1(db))$, where E is a constant query. We apply induction hypothesis and decompose $Q_1(db)$ into $P_1@Q_1^{dec}(db)$. Hence $rec_{\mathcal{S}}(E)(Q_1(db)) = rec_{\mathcal{S}}(P_1@Q_1^{dec}(db)) = rec_{\mathcal{S}}(P_1)@rec_{\mathcal{S}}(Q_1^{dec}(db))$, and we define $P \stackrel{\text{def}}{=} rec_{\mathcal{S}}(E)(P_1)$ and $Q^{dec}(db) = rec_{\mathcal{S}}(E)(Q_1^{dec}(db))$.

7. $Q(db) = (\&x := Q_1(db))$. Decompose $Q_1(db) = P_1@Q_1^{dec}(db)$ first, then $P \stackrel{\text{def}}{=} (\&x := P_1)$, $Q^{dec}(db) \stackrel{\text{def}}{=} Q_1^{dec}(db)$.

8. $Q(db) = (Q_1(db) \oplus Q_2(db))$. Decompose the sub-queries first: $Q_i(db) = P_i@Q_i^{dec}(db)$, $i = 1, 2$, then define $P \stackrel{\text{def}}{=} (P_1 \oplus P_2)$, $Q \stackrel{\text{def}}{=} (Q_1^{dec} \oplus Q_2^{dec})$.

9. $Q(db) = ()$. Take $P \stackrel{\text{def}}{=} (), Q^{dec} \stackrel{\text{def}}{=} ()$.

10. $Q(db) = \&y$ (an output marker). Take $P \stackrel{\text{def}}{=} \&, Q^{dec} \stackrel{\text{def}}{=} \&y$.

□

We can show how to evaluate any query in $\mathcal{C}$ efficiently, in the sense of Definition 1.1, on a distributed database. The algorithm is given in Fig. 15. We start by decomposing the query into $P@Q(db)$, with P constant and Q decomposed, following the procedure in Proposition 5.11, then we apply the algorithm in Fig. 15. We describe the algorithm next.

The distributed database can be expressed as:

$$db \quad = \quad \&x_1@\mathsf{cycle}_{\mathcal{X}}\,(db_1 \oplus \ldots \oplus db_m)$$

where $db_1, \ldots, db_m$ are the fragments on the m sites. Steps 1 and 2 of the algorithms compute $Q(db)$, according to the expression:

$$Q(db) \quad = \quad Q(\&x_1)@\mathsf{cycle}_{\mathcal{X}}\,(Q(db_1) \oplus \ldots \oplus Q(db_m))$$

Each server α computes $F_\alpha := Q(db_\alpha)$, for $\alpha = 1, m$. We cannot send all fragments F_α to the client, because their combined size exceeds that of the answer. Steps 3 to 7 trim the fragments F_α before sending them to the client. In step 3 each server constructs an accessibility graph for F_α. This graph is obtained by taking the input nodes and output nodes in F_α, and connecting with an edge every pair of nodes (u, v) whenever there exists a path in F_α from the input node u to the output node v. The accessibility graphs are sent to the client (Step 4): notice that their combined size is no more than $O(n^2)$, where n is the total number of cross edges. The client connects the input and output nodes of the different fragments, and forms a global accessibility graph. Next it computes the *accessible* nodes in the global accessibility graph (Step 4). To see how these are computed, consider the query's answer, $P@Q(db)$, which becomes:

$$P@Q(\&x_1)@\mathsf{cycle}_{\mathcal{X}}\,(F_1 \oplus \ldots \oplus F_m) \tag{6}$$

Recall that the global accessibility graph is a summary of $(F_1 \oplus \ldots \oplus F_m)$, showing which input nodes are connected to which output nodes. The client computes $P' \stackrel{\text{def}}{=} P@Q(\&x_1)$ first and identifies the output markers reachable in P' from its root (P' has a single input node labeled $\&$): let these markers be $\&z_1, \ldots, \&z_p$. Initially, all input nodes in the global accessibility graph labeled with these markers are marked "accessible". Then the client executes a standard accessibility algorithm to find all accessible nodes, and sends them back to the servers (Step 5). In Step 6 each server α knows the accessible input nodes for F_α, and computes the accessible local nodes in F_α. Finally these are sent back to the client which computes:

$$P'@\mathsf{cycle}_{\mathcal{X}}\,(F_1^{acc} \oplus \ldots \oplus F_m^{acc})$$

Algorithm : *Distributed-Evaluation-$\mathcal{C}$*
Input : A query of the form $P@Q(db)$
with P constant and Q decomposed,
A semistructured database db distributed on m sites:
$db = \&x_1@\mathsf{cycle}_{\mathcal{X}}\ (db_1 \oplus \ldots \oplus db_m)$
Output : Evaluates $P@Q(db)$
Method :

Step 1 Send Q to all servers α, $\alpha = 1, m$.
Step 2 At every site α compute $F_\alpha := Q(db_\alpha)$
Step 3 At every site α construct the accessibility graph from F_α (see text)
Step 4 Every site α sends it accessibility graph to the client.
The client assembles them into the global accessibility graph (see text).
then computes all nodes accessible from P's root
Step 5 Broadcast the accessible nodes to every server site α, $\alpha = 1, m$.
Step 6 Every site α computes F_α^{acc}, the accessible part of F_α.
Step 7 Every site α sends F_α^{acc} to the client
which computes the final result $P@Q(\&x_1)@\mathsf{cycle}_{\mathcal{X}}\ (F_1^{acc} \oplus \ldots \oplus F_m^{acc})$.

Figure 15: Distributed evaluation.

All nodes in $F_1^{acc}, \ldots, F_m^{acc}$ contribute to the query's answer, because they are all accessible from the answer's root: hence the total size of data exchanged in Step 7 is no larger than the query's answer. Moreover this expression is equivalent to Eq.(6), because only inaccessible nodes and edges have been eliminated from the fragments $F_1, \ldots, F_m$: hence the algorithm computes the correct answer.

In summary we have shown:

Theorem 5.12 *Every query Q in $\mathcal{C}$ can be evaluated efficiently on a distributed database db. More precisely, it can be evaluated such that:*

1. *The total number of communication steps is four (independent on the query or database).*
2. *The total amount of data exchanged during the communications is $O(n^2) + O(r)$, where n is the number of cross links and r the size of the query's result.*

Corollary 5.13 *Every restricted select-where query Q can be evaluated efficiently on a distributed database db.*

6 Distributed Evaluation of Select-Where Queries

We now turn to arbitrary select-where queries. Algorithm *Distributed-Evaluation-$\mathcal{C}$* applies only to restricted select-where queries, and we have seen in Subsection 4.1 some examples of queries that

seem more difficult to evaluate distributively than restricted select-where queries (e.g. $Q2, Q3, Q4$). We will describe here a more complex algorithm for efficient evaluation of select-where queries on distributed databases. The new algorithm uses two new techniques: *partial evaluation* and *alternating graph accessibility*. To evaluate a select-where query Q, we start by evaluating a different (but related) query Q_r: the new query is a restricted select-where query, hence we know how to compute it efficiently. We call *partial result* the result of this new query, $P \stackrel{\text{def}}{=} Q_r(db)$, and *partial evaluation* the process of computing the query in two steps (the partial result first, then the final result). After the first step, the partial result is still distributed. It contains enough information for us to reconstruct the final result, but, like in the previous distributed algorithms, it may be much larger than the real result, hence sending all its fragments to the client would violate condition 2 of Definition 1.1. The problem is now that a simple graph accessibility computation as before no longer suffices to identify the useful parts of P's fragments. The crucial observation here is that these useful parts can be computed by solving an *alternating graph accessibility problem* [Imm87, GHR95] (AGAP), which generalizes the graph accessibility problem (GAP).

We illustrate here the ideas behind both techniques with two examples. Consider the following select-where query:

$$\begin{aligned} Q(db) = \mathsf{select}\ & \{\text{“}A\text{''} \Rightarrow \{\text{“}B\text{''} \Rightarrow y \Rightarrow t_1, \text{“}C\text{''} \Rightarrow x \Rightarrow t_2\}\} \\ \mathsf{where}\ & * \Rightarrow \text{“}B\text{''} \Rightarrow x \Rightarrow t_1 \ \mathsf{in}\ db \\ & * \Rightarrow \text{“}C\text{''} \Rightarrow y \Rightarrow t_2 \ \mathsf{in}\ db \end{aligned}$$

It creates $n_1 \times n_2$ edges labeled “A'', where n_1, n_2 are the number of matchings of $* \Rightarrow$ “B''$\Rightarrow x \Rightarrow t_1$ and $* \Rightarrow$ “C''$\Rightarrow y \Rightarrow t_2$ respectively. To simplify our discussion, assume that in all possible bindings the sizes of t_1 and t_2 are $O(1)$; then the query answer has a size $O(1 + n_1 n_2)$. Notice that Q “shuffles” x, y, t_1, t_2, by grouping y with t_1 and x with t_2. Q is not a restricted select-where query because it does not use variables immediately after binding them. The variables (x, t_1) are bound first, but they cannot be used in the select clause until (y, t_2) are also bound. The idea in partial evaluation is to replace Q with a restricted select-where query Q_r:

$$\begin{aligned} Q_r(db) = \{\text{“}A\text{''} \Rightarrow\ & (\mathsf{select}\ \{\text{“}B\text{''} \Rightarrow \{x \Rightarrow t_1\}\}\ \mathsf{where}\ * \Rightarrow \text{“}B\text{''} \Rightarrow x \Rightarrow t_1\ \mathsf{in}\ db) \cup \\ & (\mathsf{select}\ \{\text{“}C\text{''} \Rightarrow \{y \Rightarrow t_2\}\}\ \mathsf{where}\ * \Rightarrow \text{“}C\text{''} \Rightarrow y \Rightarrow t_2\ \mathsf{in}\ db)\} \end{aligned}$$

Here we kept the same generators as in Q, but used each variable immediately after its binding: Q_r is a restricted select-where query, hence $Q_r(db)$ can be computed efficiently on the distributed database db. We call its answer, $P \stackrel{\text{def}}{=} Q_r(db)$, a *partial result*. P, of course, is different from the desired answer, $Q(db)$, because it contains a single edge “A'' instead of $n_1 \times n_2$, and the x's and y's are grouped differently. Still, the client can recover $Q(db)$ from P by computing another query:

$$\begin{aligned} Q(db) = Q_s(P) = \mathsf{select}\ & \{\text{“}A\text{''} \Rightarrow \{\text{“}B\text{''} \Rightarrow \{y \Rightarrow t_1\}, \text{“}C\text{''} \Rightarrow \{x \Rightarrow t_2\}\}\} \\ \mathsf{where}\ & \text{“}A\text{''} \Rightarrow \{\text{“}B\text{''} \Rightarrow x \Rightarrow t_1, \text{“}C\text{''} \Rightarrow y \Rightarrow t_2\}\ \mathsf{in}\ P \end{aligned}$$

Consider now the size of P. Since there are n_1 bindings for t_1 and n_2 bindings for t_2, the size of P is $O(1+n_1+n_2)$. The efficiency condition (Def. 1.1) allows us to send only $O(1+n_1 n_2)$ data over the network (the size of the answer, $Q(db)$). Hence, we are safe to send P with two exceptions, when $n_1 = 0$ or $n_2 = 0$ (but not both), because in this case $O(1 + n_1 + n_2)$ is larger than $O(1 + n_1 n_2)$. In these cases we should withhold sending significant portions of P. For example if $n_1 = 0$ and $n_2 > 0$, then the entire subtree under “C'' in P is useless and should not be sent. This is the purpose of the alternating graph accessibility. Consider the action taken by a server holding two

edges $u \xrightarrow{C} v \xrightarrow{y} w$, for some label y. Should it send y and w to the server ? One global condition to check is that there exists a path from the root to the node u: this ensures that (y, w) are indeed valid bindings for $* \Rightarrow$ "C" $\Rightarrow y \Rightarrow t_2$: this is similar to the previous two distributed algorithms. The other condition is to check if there exists at least one binding for (x, t_1). This is a particular case of AGAP.

To illustrate a more complex case of AGAP consider:

$$\begin{aligned} Q(db) = \textsf{select } & \{\text{“}A''\Rightarrow \{\text{“}B''\Rightarrow t_1, \text{“}C''\Rightarrow t_2, \text{“}D''\Rightarrow t_3\}\} \\ \textsf{where } & * \Rightarrow \text{“}A'' \Rightarrow t \textsf{ in } db \\ & * \Rightarrow \text{“}B'' \Rightarrow t_1 \textsf{ in } t \\ & * \Rightarrow \text{“}C'' \Rightarrow t_2 \textsf{ in } t \\ & * \Rightarrow \text{“}D'' \Rightarrow t_3 \textsf{ in } t \end{aligned}$$

which we replace with the following restricted select-where query:

$$\begin{aligned} Q_r(db) = \textsf{select } & \{\text{“}A''\Rightarrow (\textsf{select } \{\text{“}B''\Rightarrow t_1\} \textsf{ where } * \Rightarrow \text{“}B''\Rightarrow t_1 \textsf{ in } t) \cup \\ & \qquad\quad (\textsf{select } \{\text{“}C''\Rightarrow t_2\} \textsf{ where } * \Rightarrow \text{“}C''\Rightarrow t_2 \textsf{ in } t) \cup \\ & \qquad\quad (\textsf{select } \{\text{“}D''\Rightarrow t_3\} \textsf{ where } * \Rightarrow \text{“}D''\Rightarrow t_3 \textsf{ in } t)\} \\ \textsf{where } & * \Rightarrow \text{“}A'' \Rightarrow t \textsf{ in } db \end{aligned}$$

Suppose one of the servers storing a distributed database holds a node t_1 with an incoming "B''" edge. The following conditions need to be checked before it can be sent to the client. First there has to exists a path back to the root that traverses some "A''" edge. Next, from that "A''" edge there has to exists a path forward to a valid binding of t_2. Finally, from that "A''" edge there has to exists a path forward to a valid binding of t_3. Hence, instead of checking a single path (to the root), we now have to check three paths. Formalized properly, this is precisely the AGAP problem.

To summarize, given a query Q we first construct a new query $Q_r(db)$ that ignores Q's select clauses and constructs a graph containing all variable bindings, in the order with which they are introduced in the where clause. Next, before centralizing the partial result $Q_r(db)$ we have to solve an alternating graph accessibility problem that tells us which fragments are indeed useful to the final answer. We describe the details of these two steps in the remainder of this section.

6.1 Translating Select-Where Queries into Restricted Select-Where Queries

The idea behind the restricted select-where query Q_r associated to some query Q is simple. Q_r essentially collects all bindings of all variables occurring in Q, grouped in the order in which they were bound, which makes Q_r a restricted select-where query; hence Q_r can be evaluated distributively to obtain the partial result $P \stackrel{\text{def}}{=} Q_r(db)$. In a second step, we restructure P such as to obtain $Q(db)$. This restructuring doesn't need to be a restricted select-where query, since it is computed locally, at the client. It turns out that the restructuring is another select-where query Q_s, i.e. $Q(db) = Q_s(P) = Q_s(Q_r(db))$. Both Q_r and Q_s are derived from Q. We describe this step next.

Preparation Given a select-where query, we will first apply the following simple transformation to get an equivalent one:

- Reduce generators to a canonical form. Namely we replace every generator of the form $\{u_1 \Rightarrow P_1, \ldots, u_k \Rightarrow P_k\}$ in t with k generators: $u_1 \Rightarrow P_1$ in $t, \ldots, u_k \Rightarrow P_k$ in t. Similarly, we replace $u \Rightarrow v \Rightarrow P$ in t with $u \Rightarrow t'$ in $t, v \Rightarrow P$ in t', where t' is a fresh variable.

Thus, from now on, we may assume that all generators have the form $x \Rightarrow t'$ in t or $R \Rightarrow t'$ in t, with x a label variable, R a regular path expression, and t, t' tree variables.

Pattern tree Given a select-where query Q, we define a *pattern tree*, PT, that captures all dependencies between the variables used in Q. PT is obtained from the where clause of Q and the where clauses of all sub-queries. That is, PT ignores the select clauses. Nodes in PT are labeled with node variables in Q, and edges are labeled with regular path expressions or label variables. Each sub-query, Q', contributes to a certain fragment of PT that is constructed independently, then all fragments are connected by ε-edges. We describe now how to construct the fragment corresponding to some sub-query Q':

$$
\begin{aligned}
Q' = &\ \mathsf{select}\ E \\
&\quad \mathsf{where}\ a_1 \Rightarrow t_1\ \mathsf{in}\ s_1, \ldots, a_n \Rightarrow t_n\ \mathsf{in}\ s_n
\end{aligned}
$$

where a_i is either a label variable or a regular expression. Denote $V = \{t_1, \ldots, t_n, s_1, \ldots, s_n\}$ the set of all tree variables occurring in the where clause of Q' (note that $t_1, \ldots, t_n$ are distinct, but $s_1, \ldots, s_n$ may contain repeated variables, and may contain some of the t_i's). The fragment of PT corresponding to Q' has one node $n_{Q',t}$, labeled t, for every variable $t \in V$, and one edge $n_{Q',s_i} \xrightarrow{a_i} n_{Q',t_i}$ for each generator ($a_i \Rightarrow t_i$ in s_i) in Q', $i = 1, \ldots, n$. Notice that this fragment is a forest, with roots corresponding precisely to the free variables s_i in Q' (i.e. those that do not occur in $\{t_1, \ldots, t_n\}$). We call a sub-query Q' a *block*, and also call its corresponding fragment in PT a *block*. Finally, we add ε edges between nodes in different blocks, as follows. Consider again the sub-query Q' above. Its select clause has a subexpression E that may contain some other sub-query Q''. If that sub-query uses one of the variables t_i bound by Q', then we add an ε-edge from n_{Q',t_i} to n_{Q'',t_i}. Finally, the root of PT is the node $n_{Q,db}$.

Notice that if we delete all the ε edges, PT breaks into a number of connected components: we call each such component a *sub-block*. Each block in PT, corresponding to some query Q', consists of a number of sub-blocks equal to the number of free variables in the where clause of Q'.

Example 6.1 Consider the query:

$$
\begin{aligned}
Q(db) = \mathsf{select}\ &\mathsf{select}\ \{A \Rightarrow \{B \Rightarrow t, C \Rightarrow t_1, D \Rightarrow t_2\}\} \\
&\mathsf{where}\ R_1 \Rightarrow t_1\ \mathsf{in}\ t \\
&\qquad\quad R_2 \Rightarrow t_2\ \mathsf{in}\ t \\
\mathsf{where}\ &R \Rightarrow t\ \mathsf{in}\ db
\end{aligned}
$$

where R, R_1, R_2 are regular path expressions. It's associated pattern tree is shown in Figure 16. There are three sub-blocks: the topmost corresponds to the outermost select block, the lower two correspond to the inner select. □

Example 6.2 For a more complex example, consider the query:

$$
\begin{aligned}
Q = \mathsf{select}\ \{\ &\text{“}A\text{”} \Rightarrow (\ \mathsf{select}\ \{\ \text{“}AA\text{”} \Rightarrow (\ \mathsf{select}\ \{\text{“}AAA\text{”} \Rightarrow t\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{where}\ R_{11} \Rightarrow t_{11}\ \mathsf{in}\ t_1, R_{12} \Rightarrow t_{12}\ \mathsf{in}\ t_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad R_{21} \Rightarrow t_{21}\ \mathsf{in}\ t_2, R_{22} \Rightarrow t_{22}\ \mathsf{in}\ t_2)\} \\
&\qquad\qquad \mathsf{where}\ R_1 \Rightarrow t_1\ \mathsf{in}\ t, R_2 \Rightarrow t_2\ \mathsf{in}\ t), \\
&\text{“}B\text{”} \Rightarrow (\ \mathsf{select}\ \{\ \text{“}BB\text{”} \Rightarrow (\ \mathsf{select}\ \{\text{“}BBB\text{”} \Rightarrow t\}
\end{aligned}
$$

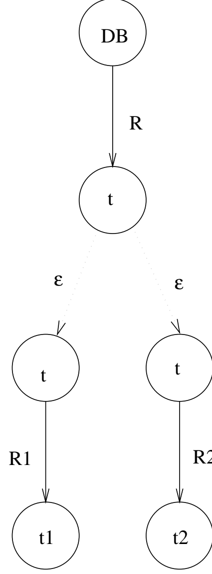


Figure 16: Pattern Tree for Example 6.1

$$\text{where } R_{31} \Rightarrow t_{31} \text{ in } t_3, R_{32} \Rightarrow t_{32} \text{ in } t_3$$
$$R_{41} \Rightarrow t_{41} \text{ in } t_4, R_{42} \Rightarrow t_{42} \text{ in } t_4)\}$$
$$\text{where } R_3 \Rightarrow t_3 \text{ in } t, R_4 \Rightarrow t_4 \text{ in } t)\}$$
$$\text{where } R \Rightarrow t \text{ in } db$$

The pattern tree is shown in Figure 17. There are five select blocks, three consisting of one sub-block, two of two sub-blocks. Hence the pattern tree has seven sub-blocks. □

Partial Result We construct the partial result P in such a way as to describe all bindings to the variables mentioned in the pattern tree. We describe its structure next. The partial result will have *match* nodes alternating with *match-set* nodes. Let $t_1, \ldots, t_n$ be the tree variables on some path starting at the root in the pattern tree (hence $t_1 = db$), and ending at some node labeled t_n. The idea is that $t_1, \ldots, t_n$ will be bound by Q in that order. Let t be the tree variable of a successor of t_n: the edge connecting t_n with t may be labeled with a label variable, say x, or with a regular expression R: i.e. there is a pattern $x \Rightarrow t$ in t_n, or $R \Rightarrow t$ in t_n in Q. So $t_1, \ldots, t_n, t$ uniquely determines a path in PT, and x labels the last edge of that path. Consider now a binding for $t_1, \ldots, t_n$ in db: intuitively, we are now about to bind t. For the given binding we introduce one *match-set* node s in P, and say that it "corresponds" to the node t in the pattern tree. Assume that there are k distinct matchings of t (and x) extending the given matchings for $t_1, \ldots, t_n$. Then s will have exactly k successors, corresponding precisely to the k bindings: $s \stackrel{\text{def}}{=} \{\text{"Match"} \Rightarrow m_1, \ldots, \text{"Match"} \Rightarrow m_k\}$, where *"Match"* is just a label constant, and each of $m_1, \ldots, m_k$ is a *matching node* (again, we say that they "correspond" to the node t in the pattern tree). Assume further that t has l successors in the pattern tree, i.e. there are patterns $y_1 \Rightarrow t'_1$ in $t, \ldots, y_l \Rightarrow t'_l$ in t; in consequence t has l successors in the pattern tree. Then each m_i has the form: $m_i = \{\text{"}x\text{"} \Rightarrow \{x\}, \text{"}t\text{"} \Rightarrow t, \text{"1"} \Rightarrow s_{i1}, \text{"2"} \Rightarrow s_{i2}, \ldots, \text{"}l\text{"} \Rightarrow s_{il}\}$, where the trees $s_{i1}, \ldots, s_{il}$ are match-set nodes for $t'_1, \ldots, t'_l$ respectively. Here "1", "2", ..., "l" are just labels representing the successors number. Of course, if there is a regular expression R instead of a label variable x, then $m_i = \{\text{"}t\text{"} \Rightarrow t, \text{"1"} \Rightarrow s_{i1}, \text{"2"} \Rightarrow s_{i2}, \ldots, \text{"}l\text{"} \Rightarrow s_{il}\}$. The intuition is the following. The partial result

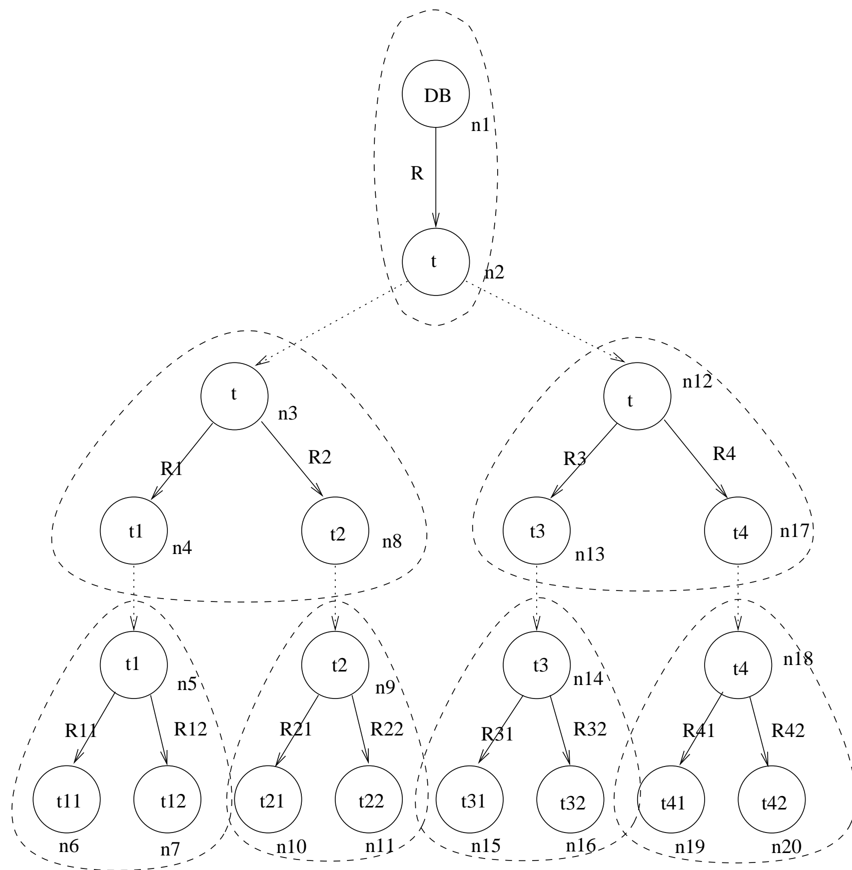


Figure 17: Pattern Tree for Example 6.2

contains all bindings of all variables in the query. For example if the pattern is $x \Rightarrow t$ in t_n, then the node m_i will have two children (labeled "x" and "t") holding the bindings of x and t respectively. The nesting structure in the partial result has been designed such that it corresponds to the order in which variables are bound in Q, making it possible to construct it with a restricted select-where query.

Partial Query No matter how complex the select-where query Q is, the partial result P can be obtained with a restricted select-where query, Q_r, which we call the *partial query*. Q_r follows the structure of the pattern tree. For each node n in the pattern tree labeled with the tree variable t we define two queries, M_n and S_n, constructing the match nodes and the match-set nodes in the partial result. The incoming edge to n can be an ε-edge, or can be labeled with a regular expression R or a label variable x. M_n has at most x and t as free variables, and is:

$$M_n(x,t) \stackrel{\text{def}}{=} \{\text{"}x\text{"} \Rightarrow \{x\}, \text{"}t\text{"} \Rightarrow t, \text{"}1\text{"} \Rightarrow S_{n_1}(t), \ldots, \text{"}k\text{"} \Rightarrow S_{n_k}(t)\}$$

Here we assume that t's successors in the pattern tree are the nodes $n_1, \ldots, n_k$ and are labeled with the tree variables $t_1, \ldots, t_k$. If n is not preceded by a label variable x, then we drop "x" $\Rightarrow \{x\}$; if n is preceded by an ε edge, then we drop "t" $\Rightarrow t$ too (because this information is stored somewhere above in P). As an optimization, we may drop any of these two components even in other cases, if they are not needed by Q's constructor (we illustrate below).

Each of the queries $S_{n_i}(t)$ constructs the match-set node of that successor node. When the edge $n \rightarrow n_i$ is an ε edge, then:

$$S_{n_i}(t) \stackrel{\text{def}}{=} M_{n_i}(t)$$

When the edge $n \rightarrow n_i$ is labeled with the regular expression R, then:

$$\begin{aligned} S_{n_i}(t) = \ & \textsf{select}\ \{\text{"}Match\text{"} \Rightarrow M_{n_i}(t_i)\} \\ & \textsf{where}\ R \Rightarrow t_i\ \textsf{in}\ t \end{aligned}$$

When the edge is labeled with the variable x_i, then:

$$\begin{aligned} S_{n_i}(t) = \ & \textsf{select}\ \{\text{"}Match\text{"} \Rightarrow M_{n_i}(x_i, t_i)\} \\ & \textsf{where}\ x_i \Rightarrow t_i\ \textsf{in}\ t \end{aligned}$$

Finally we define $Q_r(db) \stackrel{\text{def}}{=} M_{n_1}(db)$, where n_1 is the root of the pattern tree: there is no S_{n_1} query for the root node. Obviously Q_r is a restricted select-where query, because every sub-query M_n, S_n has precisely one of the forms in Def. 4.2.

Example 6.3 Consider the query Q:

$$\begin{aligned} Q(db) = \ & \textsf{select}\ \{\text{"}A\text{"} \Rightarrow \{\text{"}B\text{"} \Rightarrow t_1, \text{"}C\text{"} \Rightarrow t_2\}\} \\ & \textsf{where}\ R \Rightarrow t\ \textsf{in}\ db \\ & \qquad R_1 \Rightarrow t_1\ \textsf{in}\ t \\ & \qquad R_2 \Rightarrow t_2\ \textsf{in}\ t \end{aligned}$$

The pattern tree (not shown) has four nodes, corresponding to db, t, t_1, t_2 respectively. We denote the nodes with the variables labeling them. Then we have:

$$M_{db}(db) = \{\text{"}1\text{"} \Rightarrow S_t(db)\} \qquad // \text{ "}db\text{"} \Rightarrow db \text{ not necessary}$$

$$
\begin{aligned}
S_t(db) = \quad & \textsf{select}\ \{\ \text{“}Match\text{”} \Rightarrow M_t(t)\} \\
& \textsf{where}\ R \Rightarrow t\ \textsf{in}\ db \\
M_t(t) = \quad & \{\text{“}1'' \Rightarrow S_{t_1}(t), \text{“}2'' \Rightarrow S_{t_2}(t)\}\ //\ \text{“}t'' \Rightarrow t \text{ not necessary} \\
S_{t_1}(t) = \quad & \textsf{select}\ \{\ \text{“}Match\text{”} \Rightarrow M_{t_1}(t_1)\} \\
& \textsf{where}\ R_1 \Rightarrow t_1\ \textsf{in}\ t \\
S_{t_2}(t) = \quad & \textsf{select}\ \{\ \text{“}Match\text{”} \Rightarrow M_{t_2}(t_2)\} \\
& \textsf{where}\ R_2 \Rightarrow t_2\ \textsf{in}\ t \\
M_{t_1}(t_1) = & \{\text{“}t_1'' \Rightarrow t_1\} \\
M_{t_2}(t_2) = & \{\text{“}t_2'' \Rightarrow t_2\}
\end{aligned}
$$

We performed two optimizations (marked by comments): namely we dropped “$db'' \Rightarrow db$ and “$t'' \Rightarrow t$, since the values of db and t are not needed in the final result. Hence:

$$
\begin{aligned}
Q_r(db) = M_{db}(db) = & \\
\{\text{“}1''\Rightarrow \textsf{select}\ \{\ \text{“}Match\text{”} \Rightarrow \{\ \text{“}1''\Rightarrow\ & \textsf{select}\ \{\ \text{“}Match\text{”} \Rightarrow \{\text{“}t_1'' \Rightarrow t_1\}\} \\
& \textsf{where}\ R_1 \Rightarrow t_1\ \textsf{in}\ t, \\
\text{“}2''\Rightarrow\ & \textsf{select}\ \{\ \text{“}Match\text{”} \Rightarrow \{\text{“}t_2'' \Rightarrow t_2\}\} \\
& \textsf{where}\ R_2 \Rightarrow t_2\ \textsf{in}\ t\}\} \\
\textsf{where}\ R \Rightarrow t\ \textsf{in}\ db\} &
\end{aligned}
$$

□

Optimization Several optimizations of the partial query (and, consequently of the partial result) are possible. As explained already we may drop the “$x'' \Rightarrow \{x\}$ and/or “$t'' \Rightarrow t$ in $M_n(x,t)$. We may eliminated match nodes with a single successor, i.e. replace:

$$S_n(t) = \textsf{select}\ \{\ \text{“}Match\text{”} \Rightarrow \{\text{“}1'' \Rightarrow S_{n'}(t')\}\}\ \textsf{where}\ \ldots$$

with

$$S_n(t) = \textsf{select}\ \{\text{“}1'' \Rightarrow S_{n'}(t')\}\ \textsf{where}\ \ldots$$

Finally, we can (of course) rename the labels *“Match”, “1”, “2”*, etc. with more suggestive ones corresponding to the query. Such optimizations, including renamings, where applied to the queries Q_r illustrated at the beginning of this section.

Recovering the result from the partial result. As explained earlier, in the last step we construct the result $Q(db)$ from the partial result P by applying a restructuring query $Q_s(P)$. Q_s is obtained by modifying Q as follows. First we introduce for each node n in the pattern tree a *match variable* m_n and a *match-set variable* s_n (with one exception: there is no match-set variable for the root node). The idea is that match variables will be bound to match nodes in P, while match-set variables will be bound to match-set nodes. The match variable for the root will be P, i.e. $m_{n_1} = P$. We describe next how to construct Q_s from Q, by induction on Q's structure. Consider a sub-query $Q'(t_1, \ldots, t_k)$ of Q, where $t_1, \ldots, t_k$ are all free tree variables in the where clause. That is $Q'(t_1, \ldots, t_k)$ is:

$$
\begin{aligned}
& \textsf{select}\ E \\
& \textsf{where}\ P_1\ \textsf{in}\ s_1, \ldots, P_n\ \textsf{in}\ s_n
\end{aligned}
$$

We write n_t for the unique node labeled t in the pattern forest of Q': the roots of the pattern forest are $n_{t_1}, \ldots, n_{t_k}$. We convert Q' into a query $Q'_s(t_1, \ldots, t_k, m_{n_{t_1}}, \ldots, m_{n_{t_k}})$ as follows. (1) Every generator of the form:

$x \Rightarrow t$ in s

is changed to:

$``i" \Rightarrow s_{n_t}$ in m_s,
$``Match" \Rightarrow m_t$ in s_t
$``x" \Rightarrow x \Rightarrow _$ in m_t
$``t" \Rightarrow t$ in m_t

where i is the number of the successor of s in the pattern tree which is labeled with t. The last two generators ($``x" \Rightarrow x$ and $``t" \Rightarrow t$) may be missing, if the corresponding variables are not used in any select clause. (2) Every generators of the form:

$R \Rightarrow t$ in s

is changed to:

$``i" \Rightarrow s_{n_t}$ in m_s,
$``Match" \Rightarrow m_t$ in s_t
$``t" \Rightarrow t$ in m_t

Again, the last clause may be missing. (3) For every node n labeled with t for which we have some outgoing ε-edge $n \to n'$ in the pattern tree we introduce the following generators at the end of the generators list:

$``i" \Rightarrow ``Match" \Rightarrow m_{n'}$ in m_n

Finally, $Q_s(P)$ will be the above translation of the entire query $Q(db)$. Technically, Q_s would have two variables, $Q_s(db, P)$ (recall that P is the match variable for db, i.e. $P \equiv m_{db}$). If db does not occur in Q_s, then we are done, since Q_s now only depends on P, $Q_s(P)$. db only occurs in Q_s if it occurs as a free variable in one of Q's select clauses, like e.g. in select $\{x \Rightarrow db\}$ where $x \Rightarrow _$ in db. When this happens, we make db a bound variable by adding the clause $``db" \Rightarrow db$ in P in Q_s's select clause.

Example 6.4 Consider the query Q in Example 6.3 and its corresponding partial query Q_r. Recall that the pattern tree has four nodes, which we call (by abuse of notation) db, t, t_1, t_2. Then in Q_s we have seven additional variables, $m_{db}, s_t, m_t, s_{t_1}, m_{t_1}, s_{t_2}, m_{t_2}$ (recall that there is no match-set variable for the root db). Moreover, $m_{db} \equiv P$. Then $Q_s(P)$ is:

select $\{``A" \Rightarrow \{``B" \Rightarrow t_1, ``C" \Rightarrow t_2\}\}$
where $``1" \Rightarrow s_t$ in P // translation of $R \Rightarrow t$ in db
$``Match" \Rightarrow m_t$ in s_t
$``1" \Rightarrow s_{t_1}$ in m_t // translation of $R_1 \Rightarrow t_1 \in t$
$``Match" \Rightarrow m_{t_1}$ in s_{t_1}
$``t_1" \Rightarrow t_1$ in m_{t_1}
$``2" \Rightarrow s_{t_2}$ in m_t // translation of $R_2 \Rightarrow t_2 \in t$
$``Match" \Rightarrow m_{t_2}$ in s_{t_2}
$``t_2" \Rightarrow t_2$ in m_{t_2}

Notice that db does not occur in Q_s, hence Q_s is of the form $Q_s(P)$ and we are done. $\Box$

Summarizing, we have:

Theorem 6.5 *Given a select-where query Q, let Q_r and Q_s be the queries constructed as above. Then (1) for any database db, $Q(db) = Q_s(Q_r(db))$, and (2) $Q_r(db)$ is a restricted select-where query.*

Proof: (Sketch) Part (2) is obvious from the construction of Q_r. We only sketch here the proof for part (1), by illustrating how it works for the query in Examples 6.3 and 6.4. To keep the notations simple, we abbreviate with $E(t_1, t_2)$ the expression $\{``A'' \Rightarrow \{``B'' \Rightarrow t_1, ``C'' \Rightarrow t_2\}\}$: i.e. both $Q(db)$ and $Q_s(P)$ have the form $\mathsf{select}\ E\ \mathsf{where}\ \ldots$ We will use the notations $M_{db}, S_t, M_t, \ldots$ from Example 6.3, and recall that $Q_r(db) = M_{db}(db)$. We also rename the variables t_1, t_2 in Q_s to t'_1, t'_2, to avoid some name clashes: that is, $Q_s(P) = (\mathsf{select}\ E(t'_1, t'_2)\ \mathsf{where}\ \ldots)$. Then:

$$\begin{aligned} Q_s(Q_r(db)) = \ & \mathsf{select}\ E(t'_1, t'_2) \\ & \mathsf{where}\ ``1" \Rightarrow s_t\ \mathsf{in}\ M_{db}(db) \\ & \qquad ``\mathit{Match}" \Rightarrow m_t\ \mathsf{in}\ s_t \\ & \qquad \ldots \text{ the other generators} \end{aligned}$$

Since $M_{db} = \{``1'' \Rightarrow S_t(db)\}$, s_t will be bound to $S_t(db)$, hence:

$$\begin{aligned} Q_s(Q_r(db)) = \ & \mathsf{select}\ E(t'_1, t'_2) \\ & \mathsf{where}\ ``\mathit{Match}" \Rightarrow m_t\ \mathsf{in}\ S_t(db) \\ & \qquad \ldots \text{ the other generators} \end{aligned}$$

Now $S_t(db) = \mathsf{select}\ \{``\mathit{Match}" \Rightarrow M_t(t)\}\ \mathsf{where}\ R \Rightarrow t\ \mathsf{in}\ db$, hence m_t will be bound to $M_t(t)$ where $R \Rightarrow t\ \mathsf{in}\ db$. That is:

$$\begin{aligned} Q_s(Q_r(db)) = \ & \mathsf{select}\ E(t'_1, t'_2) \\ & \mathsf{where}\ R \Rightarrow t\ \mathsf{in}\ db \\ & \qquad ``1" \Rightarrow s_{t_1}\ \mathsf{in}\ M_t(t) \\ & \qquad ``\mathit{Match}" \Rightarrow m_{t_1}\ \mathsf{in}\ s_{t_1} \\ & \qquad ``t''_1 \Rightarrow t'_1\ \mathsf{in}\ m_{t_1} \\ & \qquad ``2" \Rightarrow s_{t_2}\ \mathsf{in}\ M_t(t) \\ & \qquad ``\mathit{Match}" \Rightarrow m_{t_2}\ \mathsf{in}\ s_{t_2} \\ & \qquad ``t''_2 \Rightarrow t'_2\ \mathsf{in}\ m_{t_2} \end{aligned}$$

Next, $M_t(t) = \{``1'' \Rightarrow S_{t_1}(t), ``2'' \Rightarrow S_{t_2}(t)\}$. Hence s_{t_1} will be bound to $S_{t_1}(t)$, and similarly for s_{t_2}:

$$\begin{aligned} Q_s(Q_r(db)) = \ & \mathsf{select}\ E(t'_1, t'_2) \\ & \mathsf{where}\ R \Rightarrow t\ \mathsf{in}\ db \\ & \qquad ``\mathit{Match}" \Rightarrow m_{t_1}\ \mathsf{in}\ S_{t_1}(t) \\ & \qquad ``t''_1 \Rightarrow t'_1\ \mathsf{in}\ m_{t_1} \\ & \qquad ``\mathit{Match}" \Rightarrow m_{t_2}\ \mathsf{in}\ S_{t_2}(t) \\ & \qquad ``t''_2 \Rightarrow t'_2\ \mathsf{in}\ m_{t_2} \end{aligned}$$

Next, $S_{t_1}(t) = \mathsf{select}\ \{ \text{“}Match\text{”} \Rightarrow M_{t_1}(t_1)\}\ \mathsf{where}\ R_1 \Rightarrow t_1\ \mathsf{in}\ t$, hence m_{t_1} will be bound to $M_{t_1}(t_1)$ where t_1 is given by $R_1 \Rightarrow t_1\ \mathsf{in}\ t$. Similarly for m_{t_2}:

$$
\begin{aligned}
Q_s(Q_r(db)) = \mathsf{select}\ & E(t_1', t_2') \\
\mathsf{where}\ & R \Rightarrow t\ \mathsf{in}\ db \\
& R_1 \Rightarrow t_1\ \mathsf{in}\ t \\
& \text{“}t_1'' \Rightarrow t_1'\ \mathsf{in}\ M_{t_1}(t_1) \\
& R_2 \Rightarrow t_2\ \mathsf{in}\ t \\
& \text{“}t_2'' \Rightarrow t_2'\ \mathsf{in}\ M_{t_2}(t_2)
\end{aligned}
$$

Finally we substitute M_{t_1} and M_{t_2} with their definition and recover $Q(db)$. □

6.2 Alternating Graph Accessibility Problem for the Partial Result

As illustrated in the example at the beginning of this section, the partial result may contain fragments which are unnecessary for the actual query result. We want to identify and delete these fragments before sending all pieces of the partial result to the client.

Recall that the partial result consists of the following components:

Match-set nodes They have links directly to match nodes.

Match nodes They have links to variable-value nodes, and to match-set nodes.

Variable-value nodes These store (i.e. have links to) the values of label variables and of tree variables.

Furthermore each match-set node and each match node “belongs” to some node in the Pattern Tree.

In addition to the accessibility problem which we had to address in the previous distributed algorithms, here there are two new reasons why fragments of the partial result may be unnecessary:

1. A non-empty sub-block of a $\mathsf{select} - \mathsf{where}$ block becomes unnecessary if some other sub-block of the same $\mathsf{select} - \mathsf{where}$ block is empty (i.e. has no matching).

2. A variable x or t bound in some $\mathsf{select} - \mathsf{where}$ block and used in a constructor in an inner block may be unnecessary if that inner block is empty.

Recall that in both algorithms discussed so far for distributed evaluation, we solve a *Graph Accessibility Problem*, GAP, on the query's result before sending it to the client. In the case of select-where queries, we have to solve an *Alternating Graph Accessibility Problem*, AGAP [Imm87, GHR95]. We review the AGAP here briefly in a form adapted to our needs.

In an AGAP we are given a graph G whose nodes are partitioned into three sets: AND nodes, OR nodes, and accessible nodes ACC: we call such a graph an AND/OR graph. We define the set of *accessible* nodes as follows:

Definition 6.6 *Given a AND/OR graph G we define the set of accessible nodes:*

1. *Any node in ACC is accessible.*

2. *Any AND node having all its successors marked accessible is accessible.*

3. *Any OR node having at least one successor marked accessible is accessible.*

The AGAP problems has as input an AND/OR graph G and a node x and asks whether x is accessible or not. It generalizes the graph accessibility problem (GAP) we had to solve earlier as follows. Recall that in that setting we were constructing the query's result, which is a rooted graph. Given a node x which is a potential node in the constructed graph, the problem was whether there exists a path from the root to x. Construct an AND/OR graph G' by reversing all edges in G and making G's root the only node in ACC. Define all other nodes to be OR nodes. Then a node x in G is accessible from the root iff it is accessible in G' according to Definition 6.6.

Returning to select-where queries, let P be the partial result produced by the query Q_r, $P = Q_r(db)$. We will construct from P an AND/OR graph G obtained by adding more nodes and edges to P. G can be constructed without any communications between sites, and has the property that a node x in P is necessary in $Q_s(P)$ iff it is accessible in G. Hence, we use G to compute all accessible nodes x.

We describe now how to construct G. It is obtained by adding one or two nodes to P, for each node in P and for each query block B. Consider one such select $-$ where block in Q, call it B. Let n be any node in the pattern tree which is in, or above a sub-block of B, and let t be the tree variable associated to n. For each match-set node or match node n' in P "belonging" to n we add a new node to P, called the *existential node* for B and n', in notation $e_{n',B}$. The intuition is that $e_{n',B}$ will tell n' whether some instantiation of the block B exists. If, in addition, n does not cover all sub-blocks of B in the pattern tree, then we add a second node to P, called the *local existential node*, in notation $le_{n',B}$. The intended meaning is that $le_{n',B}$ will be accessible iff all sub-blocks of B below n are non-empty: this depends only on the fragment of P *below* n'. By contrast $e_{n',B}$ will be accessible iff the *entire* B is nonempty: this may depend not only on information below n', but also side-wards. We show next how this accessibility information can be gathered.

Local Existential Nodes If n' is a match-set node, then $le_{n',B}$ will be an OR node, and its successors will be the le nodes of the successors of n' (if n' has any successors): intuitively, the sub-block of B dominated by n' exists (i.e. is nonempty) iff there exists at least one matching under n' for which the same sub-block exists. If n' is a match node, then we have two cases. (1) Some of n's successors in the pattern tree still dominate parts of the block B. Then $le_{n',B}$ will be an AND node, and its successors will be the le nodes of those successors which are above the block B. Intuitively, the sub-block of B dominated by n' exists if all variables following n' have a matching for which their fragment of B exists. (2) None of n's successors are above the block B: this only happens if n is a leaf in one of B's sub-blocks. Then $le_{n',B}$ is in ACC: intuitively, the fact that we have a matching node is a witness that we have instantiated that part of the block B which n' can see.

Existential Nodes We describe now the the existential nodes, $e_{n',B}$. These are always OR nodes, and their successors are constructed according to three cases. (1) n has no successors in block B, or n has some successors in the block B, but does not cover all of them. Then $e_{n',B}$ has a single successor, which is the e node of the parent of n': intuitively n' gets its information about the entire block B from some node above, which can see the entire block B. (2) n covers all nodes in B, and is the lowest node doing so: then $e_{n',B}$ has a single successor, which is $le_{n',B}$. Intuition: what the sub-block n' sees is precisely the entire block B, hence the local existential node is the same as the existential node. (3) n dominates the entire block B, and so do some of its successors: then $e_{n',B}$ has as successors the e nodes of n' successors. Intuition: there are nodes below which "know" about the non-emptiness of the entire block B.

The Data Nodes All nodes in P are imported into G as OR nodes, and all edges are imported in reversed direction: this is in the same spirit as in algorithms *Distributed-Evaluation* and

Distributed-Evaluation-$\mathcal{C}$, where for a "data node" n' we tested whether n' is accessible from the root. During copying, we make the following three changes. (1) For every match-set node n' in P "belonging" to some node n in the pattern tree which is the root of a sub-block of some block B, we make it an AND node in G with two children: one is the former parent of n' in P, the other is $e_{n',B}$. That is n' is accessible iff it is connected to P's root and the block to which it belongs exists. (2) For every edge corresponding to a variable, i.e. of the form $n'_1 \stackrel{``x''}{\rightarrow} n'_2$ or $n'_1 \stackrel{``t''}{\rightarrow} n'_2$ with x a label variable and t a tree variable, we make n'_2 into an AND node pointing to n'_1 and to a fresh node n'_3, which is an OR node pointing to all nodes of the form $e_{n_2,B}$ with B some inner block using the variable x (or t). That is we keep that variable value only if some inner block using it can be instantiated (otherwise that variable value is never used). (3) As before, we place P's old root in ACC.

Example 6.7 Consider the query of Example 6.1:

$$\begin{aligned} Q(db) = \mathsf{select}_{B'}\ & \mathsf{select}_B\ \{A \Rightarrow \{B \Rightarrow t, C \Rightarrow t_1, D \Rightarrow t_2\}\} \\ & \mathsf{where}_B\ R_1 \Rightarrow t_1\ \mathsf{in}\ t \\ & \qquad\quad\ R_2 \Rightarrow t_2\ \mathsf{in}\ t \\ \mathsf{where}_{B'}\ & R \Rightarrow t\ \mathsf{in}\ db \end{aligned}$$

Here R, R_1, R_2 are regular path expressions. There are two blocks B and B'. We consider only the inner block, B: the other one is handled in a similar way. B has two sub-blocks in the pattern tree shown in Figure 16. We show in Figure 18 (a) and (b) a (simplified) fragment of the AND/OR graph G. All continuous lines are edges imported directly from the partial result P, in reversed direction. All dotted edges are new edges in the AND/OR graph, related to the additional e and le nodes.

Examining the P subgraph first, we see that there are two matchings for the t variable: this is illustrated by the fact that the match-set node s_t has two successor match nodes, both denoted with m_t (recall that P's edges are reversed in Figure 18). Both m_t nodes have two children[9], namely the match-set nodes corresponding to t_1 and t_2 respectively. For the first match of t there are two matchings for t_1 and two for t_2. For the second match of t there is a single matching for t_1, and no matching for t_2. Part (a) of the figures illustrates the construction of the e and le nodes for the block B (those for B' are constructed in a similar manner). We describe these, starting from the bottom. On the bottom level each le node is marked ACC: intuitively this means that once we "see" a node m_{t_1}, we "know" that the first sub-block of B exists, and similarly for t_2. On the next level (with the match-set nodes s_{t_1} and s_{t_2}), each le node is an OR node. Note that of the four le nodes on this level the last one is not accessible, according to Definition 6.6, because there is no matching for t_2 there. On level further up, each le node is an AND. That is, in the scope of the variable t, the sub-blocks of B it sees exists iff both the sub-blocks for t_1 and for t_2 exists. Since t's sub-blocks of B happen to be the entire block B, here we have a link from the e node to the le node: that is the entire block B exists iff that portion seen by t exists. All other e nodes point directly or indirectly to the e nodes on this level, since here is where we have the information about the existence of the block B. In consequence, there are two candidate instantiations for the block B, corresponding to the two bindings of t. One is non-empty (the left half of the graph in Figure 18 (a)), the other is empty (the right half). The emptiness information for each block

[9]To be accurate, each m_t node would have to point to two copies of that m_t nodes, since in the pattern tree there are two successors of t both labeled t. To prevent the figure from becoming too cluttered we avoid drawing those nodes.

instantiation is then distributed to all sub-blocks. Those who find out that they belong to empty blocks do not need to be send to the client. For example the unique binding of t_1 in the right-most leave are marked inaccessible (in general there could be several such, and the savings obtained by not sending these bindings can be large).

We next describe the two ways the e nodes are used. First, each match-set node s_{t_1} and s_{t_2} on level three are AND nodes pointing to e. That is, in order for s_{t_1} to be considered "accessible", not only must it be accessible from the root in P, but the entire block B it belongs to must exist. Of the four math-set nodes on level three, the first two are accessible (left most s_{t_1}, s_{t_2}), but the last two are not (right most s_{t_1}, s_{t_2}). Note how the entire binding for the rightmost s_{t_1} is being marked inaccessible, due to the fact that there is no corresponding matching for t_2.

The second way the e nodes are used deals with the variable nodes, which we illustrate in Figure 18 (b): we simplified the figure, in order to avoid too much clutter. Here we see that each m_t node in P has a variable successor, pointing to the root of the corresponding binding of the t variable. Ultimately, we want to send that entire tree to the client, since it participates in the construction of the final result. But not all bindings are useful: in G we add, besides the reversed edge from t to m_t, a second edge from t to the e node of the block(s) where t is used. Since in our query Q the variable t is used only in the constructor of the block B, its "raison d'etre" is the existence of the block B: hence t is an AND node. In our example the first t is accessible, while the second one is not, hence the second node (and all its subsequent nodes and edges) will not be sent to the client. Here the figure is relatively simple because t is used in a single block. In general it may be used in several blocks: then we add an additional OR node, since t's raison d'etre is when at least one of those blocks exists.

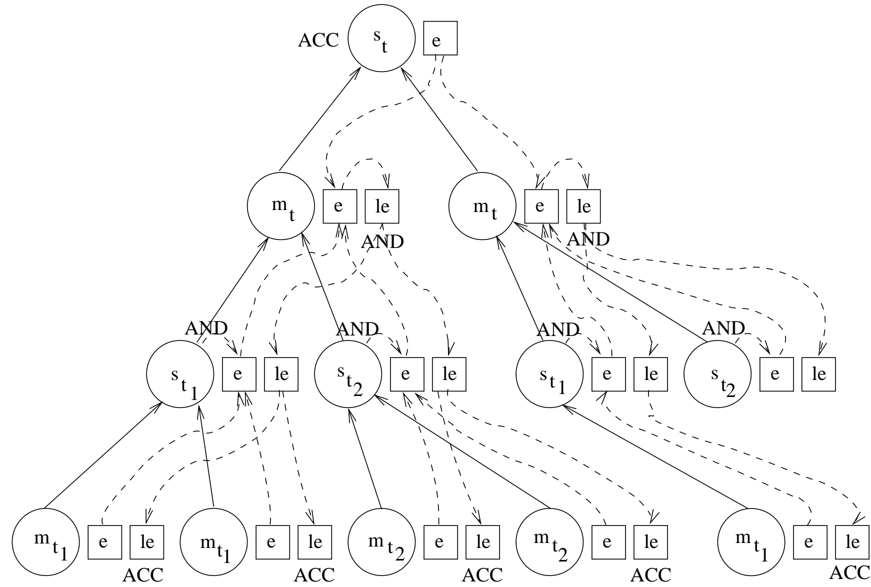
□

In general G is a graph, not a tree. However one may notice that none of G's AND nodes belongs to a cycle. We will exploit this in the next subsection where we show how to solve the AGAP distributively.

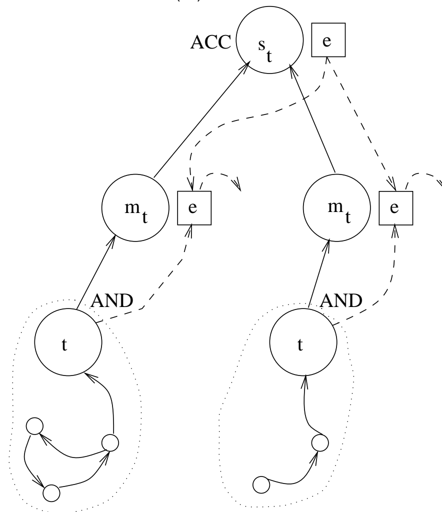
6.3 Solving the AGAP for a Distributed Database

The AGAP is inherently more difficult to solve in parallel (and, hence, distributively) than the GAP, unless $NC = PTIME$. Indeed, it is known that the GAP is in the class NC of problems computable in polylogarithmic parallel time with polynomially many processors [GHR95]. This class is widely regarded as the class of problems efficiently computable in parallel, and it is known that $NC \subseteq PTIME$, while the conjecture $NC \neq PTIME$ remains one of the major open problems in complexity theory. The AGAP problem is $PTIME$ complete with respect to NC^1 reductions [GHR95, pp.129]. Hence it is "as hard" to compute in parallel as any $PTIME$ problem, which most likely means that we would have difficulties finding an efficient distributive algorithm for a general AGAP instance.

However in our case we have to solve AGAP's of a particular form: in which the AND nodes do not belong to cycles, and in which their outdegree is "small". Specifically, we call an alternating graph G *AND-acyclic* iff none of its AND nodes belong to a cycle. Next, for a AND-acyclic graph G we define for each node n the AND-outdegree, δ_n, as follows. (1) If n and all nodes reachable from n are OR nodes or ACC nodes, then $\delta_n \stackrel{\text{def}}{=} 1$; (2) If n is an OR node with successors $n_1, n_2, \ldots$, then $\delta_n \stackrel{\text{def}}{=} \max(1, \delta_{n_1}, \delta_{n_2}, \ldots)$; (3) If n is an AND node with successors $n_1, n_2, \ldots$, then $\delta_n \stackrel{\text{def}}{=} \delta_{n_1} + \delta_{n_2} + \ldots$ Finally, given an alternating graph G we define its AND-outdegree, δ, to be the maximum AND-outdegrees of its nodes.



(a)



(b)

Figure 18: The AND/OR graph associated to a partial result. The part relevant to match and match-set nodes is in (a), that relevant to the variable nodes is in (b). Only block B is considered: there is some additional (but smaller) fragment of the AND/OR graph corresponding to the block B'.

The intuition is the follows. Consider some boolean expressions E having AND and OR operators applied to variables. This can be represented as an AND/OR tree G. Compute E's conjunctive normal form: then G's δ is the same as the largest number of operands in an AND operation in the conjunctive normal form.

First we reduce the AGAP to a GAP of exponential size:

Theorem 6.8 *Any AGAP for a graph G with n nodes can be reduced to a GAP on a graph with 2^n nodes.*

Proof: Consider the following graph G'. Its nodes are subsets of nodes of G, and there will be an edge $s_1 \rightarrow s_2$ iff for every node $n \in s_1$ either (1) $n \in s_2$, or (2) n is an OR node and some of its successor is in s_2, or (3) n is an AND node and all of its successors are in s_2. Then it is easy to check that a node x is accessible in G iff there exists a path in G' from $\{x\}$ to some s such that $s \subseteq ACC$, i.e. all nodes in s are ACC nodes. □

Corollary 6.9 *Any AGAP for a graph G with n nodes, which is AND-acyclic and has the AND-outdegree δ, can be reduced to a GAP on a graph with $\binom{n}{\delta} \leq O(n^\delta)$ nodes.*

Proof: It suffices to observe that in the graph G' of the previous theorem, if there exists a path from $\{x\}$ to some set $s \subseteq ACC$, then there exists a path going only through sets of cardinality $\leq \delta_x$. Since $\delta_x \leq \delta$, it suffices to consider in G' only nodes consisting of sets of cardinality $\leq \delta$, and there are $\binom{n}{\delta}$ such sets. □

In fact all the information about connectivity can be found in those edges $s \rightarrow s'$ of G' in which s is a singleton set. Indeed there is an edge $s \rightarrow s'$ in G' iff $\forall n \in s$ there exists an edge $\{n\} \rightarrow s''$ in G' with $s'' \subseteq s'$. This observation enables us to derive an efficient distributive algorithm for the AGAP of an AND-acyclic graph: this is shown in Figure 19. The alternating, AND-acyclic graph G is distributed on m different sites, called *servers*, and we assume to know G's AND-outdegree δ. As in previous algorithm each site starts by constructing a local graph which summarizes how inputs are connected to outputs: we call these graphs here IO_α, with $\alpha = 1, m$. The difference is now that IO_α shows for each input node x the set of sets of output nodes s' for which $\{x\}$ is connected to s' in the graph G'_α. Also, it suffices to restrict s' to sets of cardinality $\leq \delta$. In Step 2 all graphs IO_α are sent to the client, which computes all accessible input or output nodes. It then broadcasts this information to the servers, which now can compute their accessible internal nodes.

Summarizing, we have:

Theorem 6.10 *Algorithm Distributed-Evaluation-AGAP solves the AGAP for a distributed graph with the following complexities:*

1. *The total number of communication steps is constant (more exactly: two).*
2. *The total amount of data exchanged during communications is $O(n\binom{n}{\delta}) = O(n^{1+\delta})$, where n is the total number of cross links and δ the AND-outdegree.*

For the GAP problem, $\delta = 1$ and the above algorithm is essentially the accessibility computation part of Algorithm *Distributed-Evaluation* .

Finally, we apply this algorithm and the techniques describe earlier in this section, to evaluate efficiently select-where queries on distributed databases. For a select-where query Q we define its *block-fragmentation*, δ, to be the largest number of leaf nodes in any block of Q's pattern tree. For example the query in Example 6.1 has $\delta = 2$ while the query in Example 6.2 has $\delta = 4$. All restricted select-where queries have $\delta = 1$.

Algorithm : *Distributed-Evaluation-AGAP*
Input : A alternating, AND-acyclic graph G
with AND-outdegree δ
distributed on m sites, $G_\alpha, \alpha = 1, m$
Output : Computes for each node x whether x is accessible
Method :
Step 1 At every site α compute the input-output graph IO_α (see text)
Step 2 Send all IO_α, $\alpha = 1, m$ to the client.
Step 3 The client constructs the global accessibility graph
on the input/output nodes, IO
Step 4 The client broadcasts IO to all servers.
Step 5 The servers compute their accessible nodes

Figure 19: Distributed AGAP.

Theorem 6.11 *Let Q be a select-where query with b blocks and with block fragmentation δ, and db be a distributed database. Let n be the number of cross-links, and r the size of $Q(db)$. Then Q can be evaluated efficiently distributively, with the following complexity:*

1. *The total number of communication steps is four (independent on the query or data).*
2. *The total amount of data exchanged during communications is $O(n^{1+\delta})+O(r)$. The constants in the O notation depend on the query.*

Proof: (Sketch) We describe first the evaluation method which follows naturally from the techniques described in this section. The method is efficient, according to Definition 1.1, but unsatisfactory because the size of the total data exchanged is $O(n^{2+\delta}) + O(r)$. Then we show how to improve this method.

We start by decomposing the query into $Q(db) = Q_s(Q_r(db))$, where $P = Q_r(db)$ is the partial result, as before. Next we evaluate the partial result $P = Q_r(db)$ using algorithm *Distributed-Evaluation-$\mathcal{C}$* , but do not send the result to the client: instead each server α holds a fragment of the partial result, P_α, for $\alpha = 1, m$. Next we construct the associated AND/OR graph G described in Subsection 6.2. No communications are needed here, but notice that the total number of cross links has increased from n to $(2b+1)n$, where b is (recall) the total number of select $-$ where blocks in Q. We run algorithm *Distributed-Evaluation-AGAP* to compute G's accessible nodes, hence we compute at each site α the accessible part of P_α, call it P_α^{acc}. These parts are then sent to the client and assembled into P^{acc}. As our informal discussion mentioned at the beginning of this section, the size of P^{acc} is bound by $O(r)$, i.e. it is no larger than the actual result: to achieve that it was important to reduce P to P^{acc}, otherwise the size of P can be arbitrarily large when compared to r. Finally we compute $Q_s(P)$ at the client.

The above method has indeed only four communication steps, which is the same as Algorithm *Distributed-Evaluation-$\mathcal{C}$* , with the only difference that now we compute AGAP instead of GAP. However the AND-outdegree of the graph G is $1+\delta$, not δ: this results in total size of the data sent $O(n^{2+\delta})$, not $O(n^{1+\delta})$. To see what is happening, recall that each match-set node in P becomes an

AND node in G with two successors: its parent in P and the associated e node for the current block. But that e node is a large AND expression of all sub-blocks, including the fragment dominated by the current match-set node. For example in Figure 18 (a) the left-most node s_{t_1} has AND-outdegree 3, while Q's block fragmentation is 2. Intuitively s_{t_1} should be accessible if (1) it is reachable from the root AND (2) the other sub-block exists, hence it should have AND-outdegree 2. What is happening instead is that G adds a third redundant conjunct (3) s_{t_1}'s sub-block exists. That is redundant in the sense that if it is not satisfied, then s_{t_1} has no m_{t_1} successors in P, and we don't have to work to eliminate it from P^{acc}.

We can avoid this by introducing more nodes in the graph G. Instead of having a single node e for each block B, now we introduce several such nodes, one corresponding to each node in the pattern tree belonging to B: hence instead of the $e_{n',B}$ nodes, we now have $e_{n',B,n}$ nodes, where n' is some node in P, B is a block, and n is some node in Q's pattern tree, s.t. n belongs to the block B. The meaning of $e_{n',B,n}$ is that it will be accessible iff the block B exists, possible with the exception of the sub-block (in the pattern tree) dominated by n. Similarly, we construct more le nodes. The AND-outdegree of the new e and le nodes will be one less than the AND-outdegree of the e and le nodes (which we still need to keep, for the purpose of the variable nodes). Finally, in the match-set nodes we use the new e nodes, instead of the old ones. We invite the reader to fill in the details. □

7 View Maintenance

7.1 View Maintenance for Restricted Select-Where Queries

In the view maintenance problem we are given a query Q defining a view of the database, $V \stackrel{\text{def}}{=} Q(db)$. When the database is updated with an increment Δ, we want to compute the view on the updated database incrementally from Δ. By that we mean that the amount of work should depend only on the size of Δ and of V, not on that of db. In general, the increment may consist either of insertions, or deletions, or both [GL95]. In order to be able to do so, we need to store and maintain some additional information besides V.

We show here that the distributed evaluation algorithms presented in this paper can be applied to a restricted form of the view maintenance problem: namely when all updates are insertions. That is Δ consists of new nodes and new edges being added to db. Here we distinguish two cases: (1) edges are not allowed to "point back", i.e. to go from Δ into db, and (2) edges are allowed to go arbitrarily between db and Δ. The second case requires more work, because we may need to re-traverse parts of db due to the new edges entering the old graph.

The basic idea behind our view maintenance algorithms is to instantiate a distributed evaluation algorithm for Q to the case when the database is stored on two sites: site 1 holds db, while site 2 holds Δ. We keep all intermediate results at site 1, i.e. do not do any trimming of the partial result, based on knowledge about Δ. When computing the view in the first stage, $V = Q(db)$, we take $\Delta = \emptyset$. When the update actually takes place, we run the algorithm once again, but now all the processing at site 1 is already done, so we only have to process Δ. We briefly discuss this for each setting.

View Maintenance for Restricted Select-Where Queries Here we take as basis Algorithm *Distributed-Evaluation* . Consider case (1) first, when Δ is not allowed to point back. Then we run $visit_1$ in Algorithm *Distributed-Evaluation* , considering db's root to be the only input node, and all nodes as being output nodes. This results in F_1. Next we compute F_1's accessible part, F_1^{acc}: it consists of V and all pairs of the form (s, u), with u a node in db visited in state s. F_1^{acc}

will hence be our view plus the additional information consisting of such pairs (s, u). When db is updated with some Δ, we consider all edges $u \to v$ with u in db and v in Δ (the "cross edges"). For each state s such that (s, u) is in F_1^{acc}, we compute $visit_2(s, v)$. This results in new nodes and edges being added to F_1^{acc}, which updates both the view V and the additional information.

Case (2), when edges are allowed to point back, from Δ to db, is similar, but now we consider all nodes in db to be both input and output nodes for site 1. Then F_1^{acc} is much larger, since it always contains all pairs of the form (s, u): in addition there are ε edges from each (s, u) to those nodes v in db which would be included in the view V whenever u will be visited in state s. View maintenance proceeds as before.

Note that in both cases the amount of work for view maintenance is proportional only to the size of Δ, and independent on db and V.

View Maintenance for Queries in $\mathcal{C}$ Here we take as basis Proposition 5.11 and Algorithm *Distributed-Evaluation-$\mathcal{C}$* . In case (1) we add one output marker to each node in db (or only to a subset of such nodes, if we know in advance where updates are allowed to occur), and call db_1 the new database. Let $Z_1, Z_2, \ldots$ be the new output markers. We compute now $V_1 = Q(db_1)$. Not surprisingly, V_1 may be much larger than V, because all the output markers in db_1 may now be part of the result (but V_1's size is within a factor of db's size). The actual view is $V = V_1@(Z_1 := \{\}; Z_2 := \{\}; \ldots)$. Any update is now expressible as $db_1' := db_1@\Delta$. Using Proposition 5.11 we can prove that the new view, $V_1' = Q(db_1')$, can be computed as $V_1' = V_1@Q^{dec}(\Delta)$: this follows from the fact that Q can be written as $Q(db_1) = P@Q_1^{dec}(db_1)$ with Q_1^{dec} decomposable and from the associativity of @.

In case (2) we introduce both an input and output marker at each node in db: call db_1 the resulting database. We decompose Q, $Q(db) = P@Q^{dec}(db)$, and define $V_1 = Q^{dec}(db_1)$. V_1 will be even larger as before. Furthermore, an update is now $db_1' := \mathsf{cycle}\ (db_1; \Delta)$. Here Δ has an input marker for every edge from db to Δ, and an output marker for every edge from Δ to db. Since the new cross edges can form cycles, we express the update with cycle rather than[10] @. Finally we can maintain V_1 as $V_1' = Q^{dec}(\mathsf{cycle}\ (db_1; \Delta)) = \mathsf{cycle}\ (V_1; Q^{dec}(\Delta))$.

Again, in both cases the amount of work for view maintenance is proportional only to the size of Δ, and independent on db and V.

View Maintenance for Select-Where Queries Finally we consider select-where queries. Here we only consider case (1). Given a query Q, we split it into $Q(db) = Q_s(Q_r(db))$, where Q_r is the restricted select-where query computing the partial result. As before we introduce an output marker at each node in db to obtain db_1, then compute $P_1 = Q_r(db_1)$. As we know, P_1 may be larger than V, and we don't want to traverse it entirely after an update. So we compute its associated AND/OR graph G, then the graph G' (see Theorem 6.8 and Corollary 6.9). We store, besides P_1, the transitive closure of G'. If Q's block fragmentation is δ, then the transitive closure of G' is fully described by pairs of nodes $\{u\} \to s'$ for which s' is a set of cardinality $\leq \delta$ accessible from $\{u\}$. We also "simplify" these pairs, by dropping from s' all nodes which are already known to be accessible. Thus a pair $\{u\} \to s'$ means that u will become accessible in G as soon as all nodes in s' become accessible. When an update takes place, $db_1' := db_1@\Delta$, then we first update $P_1' := P_r@Q_r^{dec}(\Delta)$, where $Q_r(t) = P_r@Q_r^{dec}(t)$ is the decomposition of Q_r. Before recomputing the view however, we need to compute the accessible part of P_1'. To do this efficiently we use the stored transitive closure. Namely from $Q_r^{dec}(\Delta)$, new le nodes (see Subsection 6.2) in G' may become accessible. We consider all sets s' formed only of newly accessible nodes (their number is $O((size(\Delta))^\delta)$), and for each of

[10]When $\mathcal{X} \cap \mathcal{Y} = \mathcal{X} \cap \mathcal{Z} = \emptyset$ and $t \in DB_{\mathcal{Y}}^{\mathcal{X}}, t' \in DB_{\mathcal{Z}}^{\mathcal{Y}}$, then one can show that $t@t' = \mathsf{cycle}_{\mathcal{Y}}\ (t; t')$. Hence the update expression used in case (2) is a generalization of that used in case (1).

them mark accessible all nodes u for which $\{u\} \to s'$ was in the transitive closure of G'. Some indexing structure is required to find all u's, given an s'. This is possible since the size of s' is bound by δ, which is typically a small number. Finally, once we have the accessible part of P_1', $(P_1')^{acc}$, we compute $V' = Q_s((P_1')^{acc})$.

Unlike the previous two settings, here the amount of work done for view maintenance depends both on V and Δ.

8 Conclusions and Future Work

We have described efficient distributed query evaluation for queries on semistructured databases. The database is an edge-labeled graph and is stored on a fixed number of independent sites. All queries considered are join-free, but may contain complex combinations of patterns with regular path expressions, graph constructors, and nested sub-queries. In their most general forms, the algorithms cover two overlapping classes of queries: the class $\mathcal{C}$ (which includes queries expressed by structural recursion, including those in some core XSLT fragment), and the select-where queries (including join-free queries expressed in XML-QL and Quilt). The methods described rely on an algebraic machinery, hence they do not preclude further query optimization before evaluation at each site. All resulting distributed algorithms are *efficient*, in the sense that they do a constant number of communication steps and send an amount of data which depends only on the number of cross links and the size of the result.

We see two directions in which this work needs further extension. The first deals with joins, which our methods do not address. In the classical relational framework we have two relations R and Q stored on two distinct sites, and wish to compute $R \bowtie Q$. The standard technique uses semi-joins [KSS97], in that the join attributes from Q are sent first to the site storing R, here a semi-join is performed, and only the matching tuples are send back to Q for a join. It is not clear how to integrate this basic idea into our distributive evaluation algorithm to compute, for example, select-where queries with joins. The second direction is in connection with the ability to describe partial information about the way a database is distributed on several sites. Recent proposals [BDFS97, GW97] describe the graph's structure by another graph summarizing the nodes and edges in the database. It is possible to further annotate this graph which information about the database is distributed, and use that information in order to perform less work at each site. We believe that such techniques could further improve the distributed algorithms presented here.

References

[Abi97] Serge Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Deplhi, Greece, 1997. Springer-Verlag.

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[Aho90] Alfred V. Aho. Algorithms for finding patterns in strings. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science. Vol A: Algorithms and Complexity.* MIT Press, 1990.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley Publishing Co, 1995.

[BDFS97] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[BDS95] Peter Buneman, Susan Davidson, and Dan Suciu. Programming constructs for unstructured data. In *Proceedings of the Workshop on Database Programming Languages*, Gubbio, Italy, September 1995.

[BFS00] P. Buneman, M. Fernandez, and D. Suciu. Unql: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[Bun97] Peter Buneman. Tutorial: Semistructured data. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 117–121, 1997.

[Cat94] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, California, 1994.

[CFMR00] Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. Xml query requirements, 2000. `http://www.w3.org/TR/xmlquery-req`.

[CFR00] Don Chamberlin, Daniela Florescu, and Jonathan Robie. Quilt: an XML query language for heterogeneous data sources. In *Proceedings of WebDB*, Dallas, TX, May 2000.

[Cla99a] James Clark. XML path language (XPath), 1999. available from the W3C, `http://www.w3.org/TR/xpath`.

[Cla99b] James Clark. XSL transformations (XSLT) specification, 1999. available from the W3C, `http://www.w3.org/TR/WD-xslt`.

[Con98] World Wide Web Consortium. Extensible markup language (xml) 1.0, 1998. `http://www.w3.org/TR/REC-xml`.

[DFF$^+$98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Xml-ql: A query language for xml, 1998. `http://www.w3.org/TR/NOTE-xml-ql/`.

[DFF$^+$99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eights International World Wide Web Conference (WWW8)*, pages 77–91, Toronto, 1999.

[DGMM00] D.Calvanese, G.DeGiacomo, M.Lenzerini, and M.Vardi. Answering regular path queries using views. In *Proceedings of the International Conference on Data Engineering*, 2000.

[DMOT00] Steve DeRose, Eve Maler, David Orchard, and Ben Trafford. Xml linking language (xlink) version 1.0, 2000. `http://www.w3.org/TR/xlink`.

[FFK+97] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. STRUDEL - a web-site management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data (Systems Demonstration)*, pages 414–425, Tucson, Arizona, May 1997. System demonstration.

[FS98] Mary Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the International Conference on Data Engineering*, pages 14–23, 1998.

[GHR95] Raymond Greenlaw, H.James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation. P-Completeness Theory.* Oxford University Press, New York, Oxford, 1995.

[GL95] Timothy Griffin and Leonid Libkin. Incremental mainenance of views with duplicates. In *International Conference on Management of Data*, pages 328–339, San Jose, California, June 1995.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of Very Large Data Bases*, pages 436–445, September 1997.

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

[JJM92] J.L.Balcazar, J.Gabarro, and M.Santha. Deciding bisimilarity is P-complete. *Formal Aspects of Computing*, 4(6A), 1992.

[KS95] David Konopnicki and Oded Shmueli. Draft of W3QS: a query system for the World-Wide Web. In *Proc. of VLDB*, 1995.

[KSS97] Henry F. Korth, Abraham Silberschatz, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, New York, 1997.

[Lyn97] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufman, 1997.

[Mil89] Robin Milner. *Communication and concurrency.* Prentice Hall, 1989.

[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.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *International Conference on Deductive and Object Oriented Databases*, pages 319–344, 1995.

[Rob99] Jonathan Robie. The design of xql, 1999. `http://www.texcel.no/whitepapers/xql-design.html`.

[Suc96] Dan Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *Proceedings of the International Conference on Very Large Data Bases*, pages 227–238, September 1996.

[TMD92] J. Thierry-Mieg and R. Durbin. Syntactic Definitions for the ACEDB Data Base Manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge,CB2 2QH, UK, 1992.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of 14th ACM SIGACT Symposium on the Theory of Computing*, pages 137–146, San Francisco, California, 1982.

[VK88] Patrick Valduriez and Setrag Khoshafian. Parallel evaluation of the transitive closure of a database relation. *International Journal of Parallel Programming*, 17(1):19–42, 1988.