Distributed Query Evaluation on Semistructured Data

Dan Suciu[*]

Abstract

Various query language have been proposed lately for querying *semistructured data*, whose data model is that of 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 subqueries. An often cited example of semistructured data is that of web sites, where the pages correspond to nodes in the graph, and hyperlinks correspond to the labeled edges: a query with a regular path expression traverses all or part of that graph.

In this work we address the problem of efficient query evaluation on distributed, semistructured databases. In our setting the nodes of the database are stored at a fixed number of sites, and the edges can be either local (with both ends in the same site), or 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* [BDHS96a] which in addition to regular path expressions can perform non-trivial restructuring of the graph. And third, for a class of queries we call *selection queries* which combine several regular path expressions with data restructuring and subqueries.

Finally, we show how these methods can be used to derive efficient view maintenance algorithms.

1 Introduction

Semistructured data is generally described as data which does not conform to a rigid schema [Abi97, Bun97]. Data components can be missing, can be of different type from one item to another, can be atomic in one item and structured in another, collections can be heterogeneous, etc [QRS+95]. There have been several motivations for considering semistructured data: data integration [PGMW95], biological databases [BDHS96a], querying or managing web sites [MMM96, KS95, FFK+97], or dealing with semi-structured data directly [QRS+95].

The various datamodels proposed for semistructured data are similar, with minor variations: they model a database as a rooted, labeled graph. The nodes 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 are attached to the graph's edges. Figure 1 illustrates an example of such a semistructured databases: a fragment of the web site as `http://www.ucsd.edu`.

Several languages have been proposed for semistructured data, which vary in style and expressive power. At their core, all share a common feature: that of describing in a declarative way the

[*]AT&T Laboratories — Research, `suciu@research.att.com`

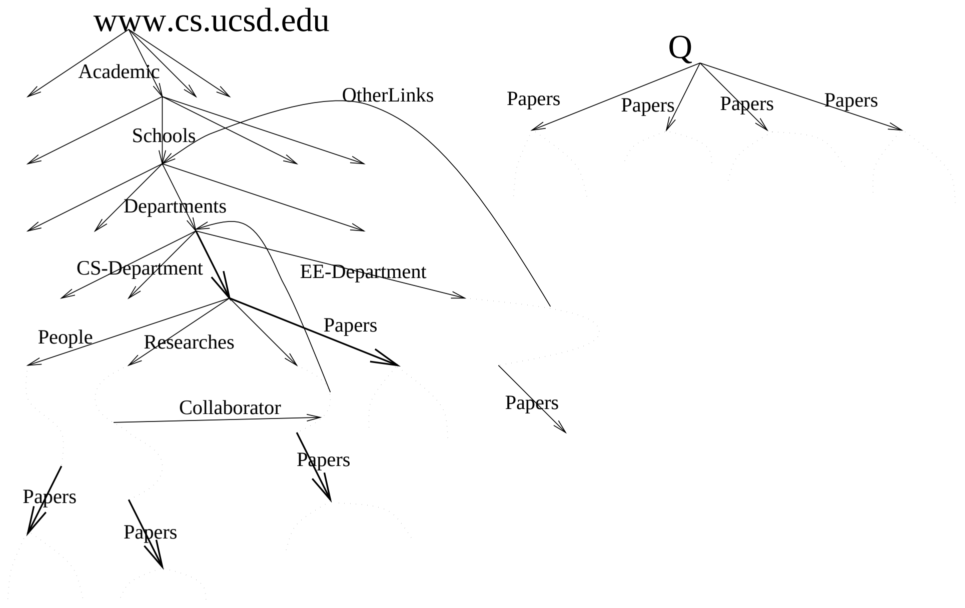


Figure 1: A fragment of the web site at `http://www.ucsd.edu`, and the result of a query Q. For the purpose of structure specific queries such a data source can be modeled as an edge labeled tree.

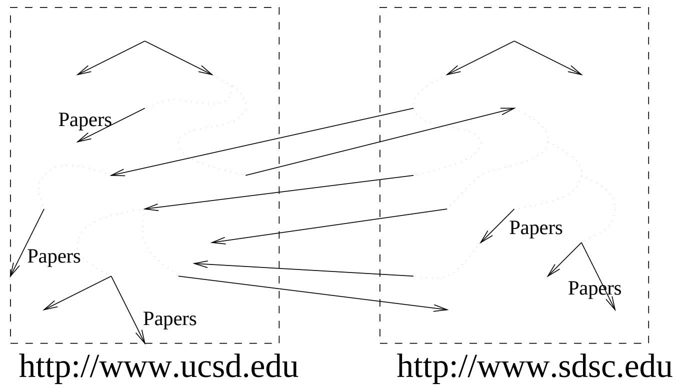


Figure 2: The information source is distributed on two sites.

traversal paths based on regular path expressions. For example the following UnQL [BDHS96a] query:

$$Q1 = \textsf{select } t$$
$$\textsf{where } * \Rightarrow \textit{“CS-Department”} \Rightarrow * \Rightarrow \textit{“Papers”} \Rightarrow t \textsf{ in } DB$$

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.

So far, systems managing semistructured data evaluate regular path expressions in a naive way, by searching the graph from the root. Some heuristic-based optimizations are discussed in [BDHS96a], and some data structures for optimizations are introduced in [BDFS97, GW97].

In this work we address the problem of efficient query evaluation on distributed, semistructured databases. 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). In general there can be more than two sites, but we assume that the number of sites is known in advance. A naive evaluation of the previous query on this distributed database would require us to ship the query between the two sites back and forth a number of times dependent on the number of cross links. Since communications are much more expensive than local computations, this solution is unacceptable.

We propose here *efficient* distributed query evaluation methods. By efficient we mean:

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 total number of cross edges, and (b) the size of the total result.*

The naive evaluation of a regular path expression would violate item 1 above. Another naive evaluation would be to send the entire database to a single site in one communication step (a "gather" step), then compute the query locally: but this would violate 2, since the size of the database is much larger than the number of cross edges or of the size of the query's result.

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, BDHS96a]. This allows us to define a query as a collection of mutually recursive functions which iterate on the graph's structure. The queries in this formalism form an algebra $\mathcal{C}$, which is a fragment of the algebra UnCAL [BDHS96a]: $\mathcal{C}$ is weaker than UnCAL, because it doesn't have joins and uses the weaker form of structural recursion. Still, all regular queries can be expressed in $\mathcal{C}$, and in addition $\mathcal{C}$ can express complex graph restructuring queries. For this framework we develop an algebraic approach to distributed query evaluation, rather than an operational one. Namely for each query Q in $\mathcal{C}$ 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 recent work [BDFS97, GW97] has addressed the problem of using certain knowledge of the structure of the database in order to optimize queries with regular path expressions. If such knowledge is available about the database fragment stored at some sites, then these techniques could be used to optimize the evaluation of Q^{dec} at those sites. For the correctness of this algebraic method we rely on the algebraic machinery developed in [BDHS96a, BDHS96b]: 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 freely several regular expressions with existential conditions, implicit cartesian products, various forms of restructurings, and nested select $-$ where queries. These queries form a fragment of UnQL [BDHS96a], and we call them *selection queries*. We describe an efficient distributed evaluation algorithm for all join-free selection 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 selection 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 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; 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 [BDHS96a, BDHS96b]. We define regular queries and give an efficient distributed evaluation algorithm in Section 3. In Section 4 we define more general selection queries, and explain why their distributed evaluation is more difficult than for regular path expressions. We define the calculus $\mathcal{C}$ in Section 5, present the algebraic method for distributed evaluation of $\mathcal{C}$ queries, and prove its correctness. We describe efficient distributive evaluation of arbitrary selection 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

There seems to be a consensus lately to model semistructured as labeled graphs: see [Abi97, Bun97]. The different proposals considered have only minor differences. Since we focus on the language UnQL we adopt the data model proposed in [BDS95, BDHS96a]. It has two features not present in other data models [PGMW95, QRS+95], markers and ε-edges, which allow us to describe easier our distributed evaluation algorithms.

2.1 Rooted, Labeled Graphs

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

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

In our query language we can test the type of a given label a, with predicates like $\mathit{isInt}(a)$, $\mathit{isReal}(a)$, $\mathit{isImage}(a)$, 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 $\mathit{Label} \cup \{\varepsilon\}$ — 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 Tree 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 hyperlinks. 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 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: 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: we will argue in the sequel that the unaccessible parts of a rooted graph may be deleted.

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, etc.

2.2 Syntax

Following [BDHS96a] we use a concrete syntax for denoting a particular case of semistructured data: data whose underlying graph is a tree. The syntax is:

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

r1:

m	n
a	c
b	c
c	b

r2:

m	p	q
b	c	c
a	a	c

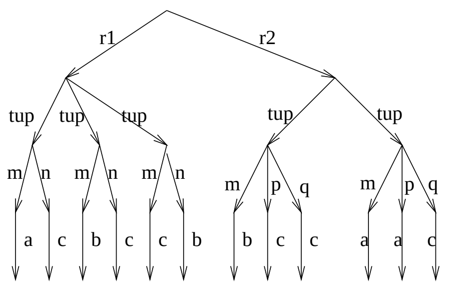


Figure 3: A relational database represented as a tree.

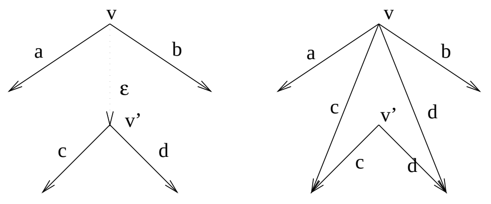


Figure 4: An ε edge from node v to v' means that all edges "visible" from v' should be "visible" from v too: the second tree is equivalent to the first one. Note that, if no other edge points to v', then v' may be eliminated, since it is no longer accessible from the root.

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:

$$
\begin{aligned}
\{r1 \Rightarrow \{&tup \Rightarrow \{m \Rightarrow \{a\}, n \Rightarrow \{c\}\},\\
&tup \Rightarrow \{m \Rightarrow \{b\}, n \Rightarrow \{c\}\},\\
&tup \Rightarrow \{m \Rightarrow \{c\}, n \Rightarrow \{b\}\}\},\\
r2 \Rightarrow \{&tup \Rightarrow \{m \Rightarrow \{b\}, p \Rightarrow \{c\}, q \Rightarrow \{c\}\},\\
&tup \Rightarrow \{m \Rightarrow \{a\}, p \Rightarrow \{a\}, q \Rightarrow \{c\}\}\}\}
\end{aligned}
$$

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 the sequel.

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 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 exaplain and construct than (b), since the latter involves "merging" of two

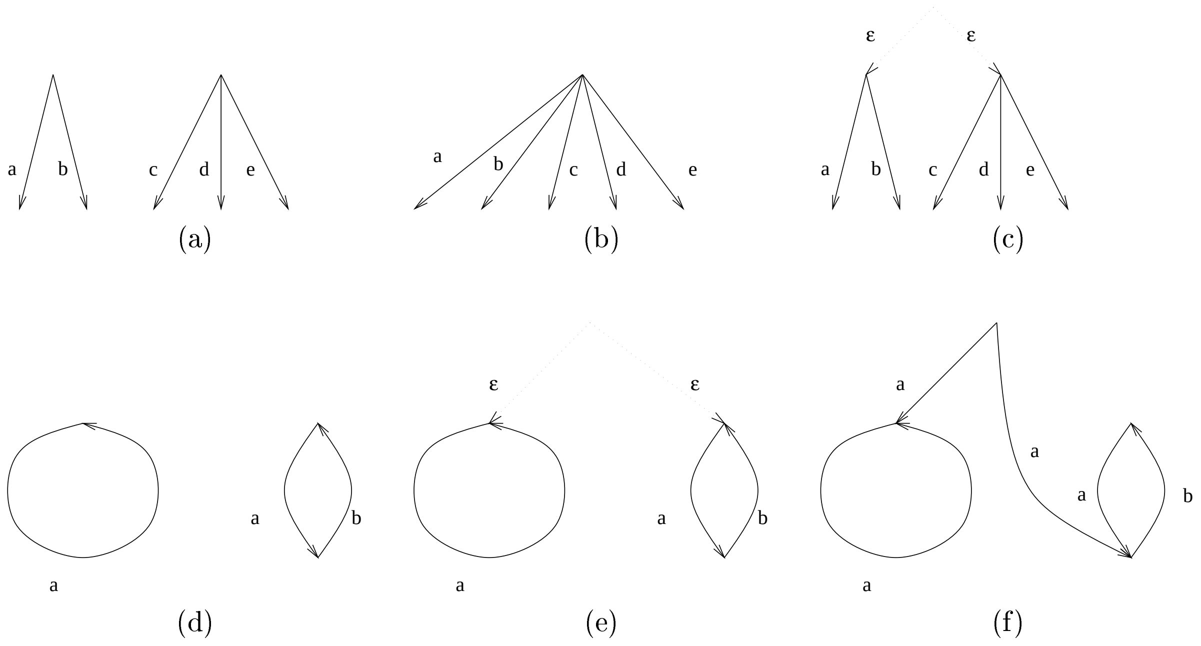


Figure 5: Illustration of how ε-edges can be used to express union.

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, BDHS96a] we say that two graphs are equal if they are *bisimilar*. For completeness, we give the formal definition in Appendix A. Adapting bisimulation as graph equality has three important consequences. First, on encodings of sets bisimulation coincides with set equality, hence 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 bismilar to the original one. Third, every rooted graph is bisimilar to its accessible part.

The reader may skip the formal definition in Appendix A and consider the following intuitive (but not very formal) test for checking bisimulation between two graphs: G and G' are bisimilar iff they become equal after (1) unfolding into (possible infinite) trees, and (2) duplicate elimination at each node.

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:

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

Here P is any unary, user-defined predicate on *Label*, or boolean combinations of such predicates, and $a \in \mathit{Label}$ is any label constant[1]. The expression $\_$ denotes any label[2], $R_1|R_2$ 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:

$$\_* \Rightarrow \mathit{CSDept} \Rightarrow \_* \Rightarrow \mathit{Paper}$$

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 $\mathit{Paper}(x) \overset{\text{def}}{=} (x = \text{“}\mathit{Publication}\text{”} \text{ or } x = \text{“}\mathit{Paper}\text{”} \text{ or } x = \text{“}\mathit{Technical\ Report}\text{”})$.

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:

$$\_* \Rightarrow \mathit{CSDept} \Rightarrow (\mathit{not}(\mathit{Dept}))* \Rightarrow \mathit{Paper}$$

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

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

□

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

We use regular expressions in queries of the form:

$$\begin{aligned} Q(DB) = {} & \textsf{select } t \\ & \quad \textsf{where } R \Rightarrow t \textsf{ in } DB \end{aligned}$$

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*.

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. First we identify a node u in DB with the labeled graph having the same body as DB and root u. Let $t_1, \ldots, t_n$ be all the labeled graphs corresponding to nodes in DB matching the regular path expression R. Then the meaning of Q is: $t_1 \cup \ldots \cup t_n$. To evaluate Q it suffices to first find all nodes $u_1, \ldots, u_n$ in DB reachable from the root through a path matching R, then create a new node ρ and add n ε-edges from ρ into $u_1, \ldots, u_n$.

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

[2]Thus $\_$ is the same as the predicate true.

For an example, consider the query:

$\textsf{select}\ t$

$\textsf{where}\ * \Rightarrow \mathit{CSDept} \Rightarrow (\mathit{not}(\mathit{Dept}))* \Rightarrow \mathit{Paper} \Rightarrow t\ \textsf{in}\ DB$

and the database in Figure 9 (a): the result is in depicted in (b), and in (c) (with ε edges remored). 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:

$\textsf{select}\ \{``\mathit{Result}'' \Rightarrow t\}$

$\textsf{where}\ * \Rightarrow \mathit{CSDept} \Rightarrow (\mathit{not}(\mathit{Dept}))* \Rightarrow \mathit{Paper} \Rightarrow t\ \textsf{in}\ DB$

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 be the automaton associated to the regular expression R. Assume A has k states, $\mathit{States}(A) = \{s_1, \ldots, s_k\}$, 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 $\mathit{visit}(s, u)$ in a data structure $\mathit{result}[s, u]$. 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 $\rho \xrightarrow{\varepsilon} u_1, \ldots, \rho \xrightarrow{\varepsilon} u_n$.

Now assume that the database is distributed. A direct execution of the algorithm in Figure 6 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.

We describe next an efficient, distributed evaluation algorithm. We asume the database DB to be distributed on m sites denoted s_α, $\alpha = 1, m$. We call these sites *servers*. 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: this is not really necessary, but simplifies the exposition. 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.

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:

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

```
$Algorithm$ : $Basic\text{-}Evaluation$
$Input$ :     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 $\rho$ 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

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, $\mathit{InputNodes}(DB_\alpha)$, $\mathit{OutputNodes}(DB_\alpha)$. Whether this assumption is realistic or not depends on the way 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, $r \in Nodes(DB_1)$.

Figure 7 describes the distributed evaluation algorithm. 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_α. First, when visit_α 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_α is called on u with all states $s \in \mathit{States}(A)$ (Step 2). Second, when visit_α 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 $u \in \mathit{OutputNodes}(DB_\alpha), v \in \mathit{InputNodes}(DB_\beta), s \in \mathit{States}(A)$), 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 crosslinks, and computes all nodes accessible from (s_1, r). In Step 5 it sends these nodes back to the servers. At this point 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 3.2 *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 3.3 Consider the following regular query:

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

```
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 $\alpha$, $\alpha = 1, m$.
            Step 2  At every site $\alpha$ construct $F_\alpha$ as follows:
                    Include all nodes and arcs from $DB_\alpha$ into $F_\alpha$.
                    $visited_\alpha \leftarrow \{\}$
                    forall $r \in InputNodes(DB_\alpha), s \in States(A)$ do
                        $S \leftarrow visit_\alpha(s, r)$
                        create a new node $(s, r)$ in $F_\alpha$
                        forall $p \in S$ do
                            Add a new edge $(s, r) \xrightarrow{\varepsilon} p$ to $F_\alpha$
            Step 3  At every site $\alpha$ construct the accessibility graph for $F_\alpha$ (see text)
            Step 4  Every site $\alpha$ 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 $\alpha$, $\alpha = 1, m$.
            Step 6  Every site $\alpha$ computes $F_\alpha^{acc}$, the accessible part of $F_\alpha$.
            Step 7  Every site $\alpha$ sends $F_\alpha^{acc}$ to the client site,
                    where it is assembled into the result.
```

```
function $visit_\alpha(s, u)$
  if $(s, u) \in visited_\alpha$ then return $result_\alpha[s, u]$
  $visited_\alpha \leftarrow visited_\alpha \cup \{(s, u)\}$
  $result_\alpha[s, u] \leftarrow \{\}$
  if $s$ is a terminal state then $result_\alpha[s, u] \leftarrow \{u\}$
  if $u$ is an output node
    then create a new node $(s, u)$ in $F_\alpha$
         $result_\alpha[s, u] \leftarrow result_\alpha[s, u] \cup \{(s, u)\}$
  forall $u \xrightarrow{a} v$ (* edge in $DB$ *) do
    if $a = \varepsilon$ then $result_\alpha[s, u] \leftarrow result_\alpha[s, u] \cup visit_\alpha(s, v)$
    else forall $s \xrightarrow{P} s'$ (* automata transition *) do
            if $P(a)$ then $result_\alpha[s, u] \leftarrow result_\alpha[s, u] \cup visit_\alpha(s', v)$
  return $result_\alpha[s, u]$
```

Figure 7: Distributed Evaluation Algorithm for Regular Queries

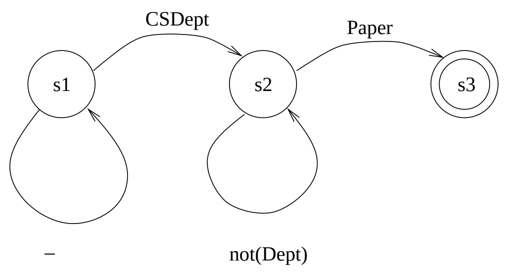


Figure 8: Automaton for Example 3.3

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, F_2). For example F_1 has an ε edge from (s_2, v_4') to u_1 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. 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_1, v_2') accessible too (recall that the client completes the missing ε cross edges). Hence (s_2, v_4), then v_3 and (s_3, v_3) are found accessible. Finally (s_3, u_2) (since there is a ε edge from v_3 to u_2), then u_2 are found 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 Selection Queries

Most applications require more than just regular path expressions. We discuss here a class of queries which we call *selection 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 [BDHS96a] whose syntax was inspired from OQL [Cat94].

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

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

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

select E
where $C_1, \ldots, C_n$

Here $C_1, \ldots, C_n$ are *conditions*, and can be of two kinds. The first kind is a *generator* and has the form: P in t, with P a pattern and t a tree variable. The second is a predicate applied

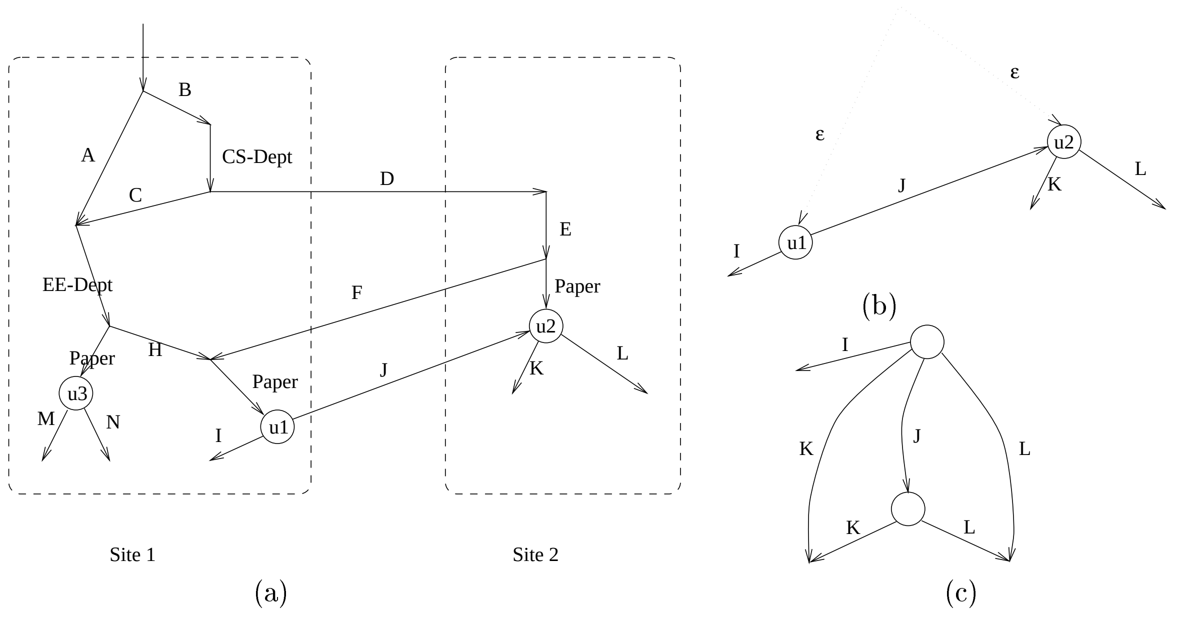


Figure 9: A database and the result of the regular path expression $* \Rightarrow$ *"CSDept"* $\Rightarrow$ *not*(*"Dept"*) $*$ $\Rightarrow$ *"Paper"*.

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 selection 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:

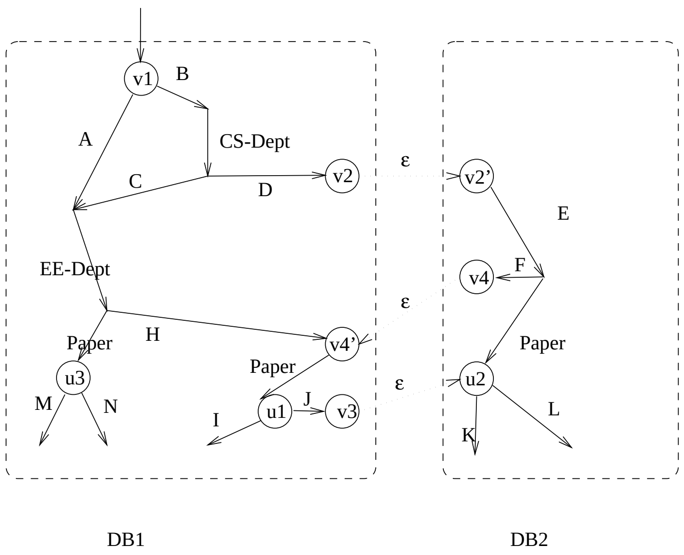
$$\begin{aligned}&\textsf{select}\ \{\text{“}Result'' \Rightarrow \{\text{“}Paper'' \Rightarrow t_1, \text{“}Abstract \Rightarrow t_2\}\}\\&\textsf{where}\ * \Rightarrow CSDept \Rightarrow not(Dept) * \Rightarrow Paper \Rightarrow t_1\ \textsf{in}\ DB\\&\qquad * \Rightarrow \text{“}Abstract\text{”} \Rightarrow t_2\ \textsf{in}\ t_1\end{aligned}$$

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. 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[3] and we drop it from our select expressions.

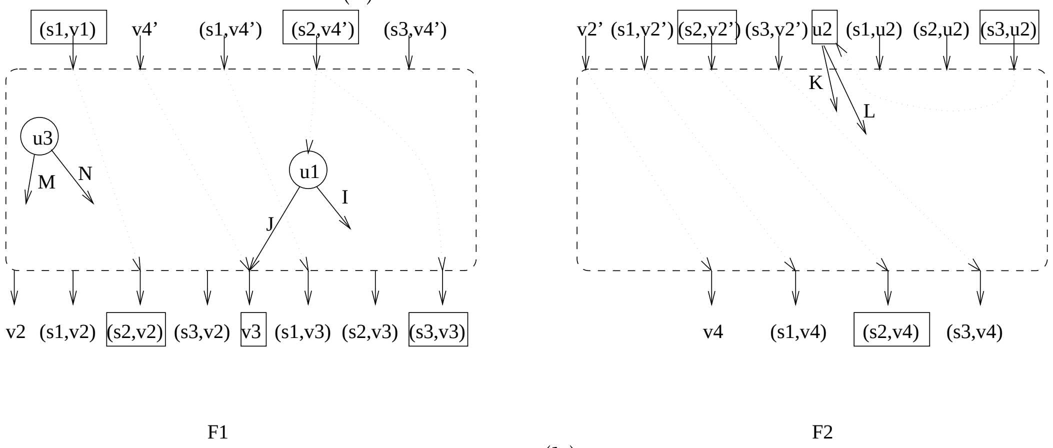
The semantics of a select query is the standard, active domain semantics. Given a query $Q =$ select E where $C_1, \ldots, C_n$, define a *valuation* to be a mapping θ sending label variables to labels and tree variables to rooted graphs consisting of the same graph as DB but with possibly different roots[4], such that all conditions $C_1, \ldots, C_n$ are satisfied. Let $\theta_1, \ldots, \theta_m$ be all valuations

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

[4]This is the same as saying that a tree variable is mapped to a node in DB.



(a)



(b)

Figure 10: Example 3.3 continued.

of a query Q. Then its meaning is $E[\theta_1] \cup \ldots \cup E[\theta_m]$.

Selection queries satisfy the following two identities which we will need in the sequel (see [BDHS96b] for more identities):

$$
\begin{aligned}
\mathsf{select}\ E\ \mathsf{where}\ C_1, \ldots, C_n &= \mathsf{select}\ (\mathsf{select}\ E\ \mathsf{where}\ C_{i+1}, \ldots, C_n)\ \mathsf{where}\ C_1, \ldots, C_i & (1)\\
\mathsf{select}\ (E \cup E')\ \mathsf{where}\ C_1, \ldots, C_n &= (\mathsf{select}\ E\ \mathsf{where}\ C_1, \ldots, C_n) \cup (\mathsf{select}\ E'\ \mathsf{where}\ C_1, \ldots, C_n) & (2)
\end{aligned}
$$

Selection 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 [BDHS96a]:

Proposition 4.1 *Let Q be a selection 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 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 selection query.*

4.1 Difficulties in Distributed Evaluation of Selection Queries

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

Data Restructuring with Grouping Consider the following query:

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

In essence this query computes the regular path expression $* \Rightarrow$ *“Paper”* $\Rightarrow x \Rightarrow * \Rightarrow$ *“Title”* but a certain edge label, x, is used in constructing the result. For a distributed database, the 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:

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

The difference between this query and the previous one is the way the x's are grouped: here, the same x will be copied several times, once for every *“Title”* accessible from it. 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:

$$Q3 = \mathsf{select}\ t$$
$$\mathsf{where}\ * \Rightarrow \text{“}Paper\text{”} \Rightarrow t\ \mathsf{in}\ DB$$
$$* \Rightarrow \text{“}Abstract\text{”} \Rightarrow \_\ \mathsf{in}\ t$$

Here the relevant regular expression is $* \Rightarrow$ *“Paper”* $\Rightarrow t$ but there is an additional filtering of only those papers which have an *“Abstract”*. 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:

$$Q4 = \mathsf{select}\ \{x \Rightarrow \{y \Rightarrow t_1, z \Rightarrow t_2\}\}$$
$$\mathsf{where}\ * \Rightarrow \text{“}Paper\text{”} \Rightarrow x \Rightarrow t\ \mathsf{in}\ DB$$
$$* \Rightarrow \text{“}Title\text{”} \Rightarrow y \Rightarrow t_1\ \mathsf{in}\ t$$
$$* \Rightarrow \text{“}Abstract\text{”} \Rightarrow z \Rightarrow t_2\ \mathsf{in}\ t$$

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 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.

$$\mathsf{select}\ \{\text{“}tup\text{''} \Rightarrow \{\text{“}m\text{''} \Rightarrow \{m \Rightarrow t_1\}, \text{“}n\text{''} \Rightarrow \{n \Rightarrow t_2\}, \text{“}p\text{''} \Rightarrow \{p \Rightarrow t_3\}, \text{“}q\text{''} \Rightarrow \{q \Rightarrow t_4\}\}\}$$
$$\mathsf{where}\ \text{“}r1\text{''} \Rightarrow \text{“}tup\text{''} \Rightarrow \{\text{“}m\text{''} \Rightarrow m \Rightarrow t_1, \text{“}n\text{''} \Rightarrow n \Rightarrow t_2\}\ \mathsf{in}\ DB,$$
$$\text{“}r2\text{''} \Rightarrow \text{“}tup\text{''} \Rightarrow \{\text{“}n\text{''} \Rightarrow n' \Rightarrow t_2', \text{“}p\text{''} \Rightarrow p \Rightarrow t_3, \text{“}q\text{''} \Rightarrow q \Rightarrow t_4\}\ \mathsf{in}\ DB,$$
$$n = n'$$

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 Extended Regular Queries

As the previous subsection suggested, some selection 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 subqueries. Subqueries 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, subqueries may have different inputs.

Definition 4.2 *An extended regular 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 extended regular queries with variables marked accordingly.

The intuition is the following. If a query select E where P in DB is extended regular, 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 subqueries, except that t may be used in immediate subqueries like select E' where P' in t. The same property holds recursively, for the subqueries.

All regular queries are extended regular queries. Query $Q1$ of Subsection 4.1 is an extended regular query, but queries $Q2, Q3, Q4$ are not. To see a more complex extended regular 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
```

Selection queries with several generators, like select E where $C_1, \ldots, C_n$ may be equivalent to extended regular 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 Selection Queries

We will only consider join-free queries. From Section 5 it will follow that extended regular queries can be evaluated efficiently on distributed databases, essentially in the same way as regular queries. Then we show in Section 6 how all join-free selection queries can be evaluated efficiently, using a new algorithm.

5 An Algebraic Approach to Distributed Evaluation

Algorithm *Distributed-Evaluation* of Subsection 3.2 works only for regular queries, which form only a small subset of the selection queries. It also has drawback of being procedural: this prohibits any further optimizations at the local sites.

We show here how to exploit UnQL's algebraic foundations [BDS95, BDHS96a, BDHS96b] in order to derive an algebraic approach to distributed query evaluation. Given a query Q we derive a *decomposed* query Q^{dec}: to evaluate Q it suffices to evaluate Q^{dec} independently on all fragments of a distributed database, then to assemble the separate pieces of the result at the client site. We call Q^{dec} a *decomposed* query. Intuitively the relationship between Q and Q^{dec} is the same as that between Steps 2 of algorithms *Basic-Evaluation* and *Distributed-Evaluation* : Q^{dec} computes slightly more than Q, because it doesn't know how a particular fragment relates to the rest of the database.

Our main result in this section consist in showing that every positive, join-free query Q in a certain calculus in [BDHS96a] can be *decomposed*, i.e. the query Q^{dec} exists. Such queries include all regular queries and extended regular queries, but also more complex ones, which perform more complicated restructurings. They do not include however all selection queries: for these we will derive a more complex algorithm in Section 6.

When applied solely to the selection queries of Section 4 this result only tells us how to evaluate distributively extended regular queries. The reader may wonder why we develop all the algebraic formalism in this section, instead of extending algorithm *Distributed-Evaluation* to extended regular queries directly. There are three reasons for doing so. First the method described here is *algebraic*: Q^{dec} is still a query and as a consequence one can apply additional optimizations to Q^{dec}, which could be even tailored specifically to each site. For example recent work [BDFS97, GW97] has addressed the problem of using certain knowledge of the structure of the database in order to optimize queries with regular path expressions. If such knowledge is available about the database fragments stored at some sites, then these techniques could be used to optimize the evaluation of Q^{dec} at those sites. Second, as we said, this method applies also to queries performing more complex restructurings than the extended regular queries. This means that it can be applied to other query languages for semistructured data. One such example is StruQL, the query language at the core of the web site management system Strudel [BDS95, FFK$^+$97]. Some of the queries in StruQL are more complex than extended regular queries, yet they can be decomposed according to the method shown here. Finally we believe that this algebraic approach to distributed query evaluation is a good illustration of the power of the algebraic foundation presented in [BDHS96a, BDHS96b].

We start by revising the calculus described in [BDHS96a, BDHS96b], showing that, in particular, it expresses all extended regular queries. Next we state and prove our main result on query decomposition.

5.1 Composing Graphs

Graphs with Multiple Inputs and Outputs In Subsection 3.2 we worked with fragments of the database DB stored at each site α, and called it DB_α. It was crucial to identify input and output nodes in each fragment DB_α, in order to describe how the fragments were put together.

Here we formalize this idea, using the algebraic formalism in [BDS95, BDHS96a, BDHS96b]. We define a labeled graph G with m inputs and n outputs to be a labeled graph as in Subsection 2.1, but in which m nodes have been designated as input nodes and n nodes are designated output nodes. Keeping in mind the intuition from Subsection 3.2, we require the output nodes to be leaves in G (i.e. have no outgoing edges). In particular the rooted, labeled graphs defined in Subsection 2.1 have 1 input and 0 outputs.

We label both input and output nodes with special symbols, called *markers*: when the input nodes are labeled with markers $X_1, \ldots, X_m$ and the output nodes with $Y_1, \ldots, Y_n$ respectively, then we say that G has inputs $\mathcal{X} \stackrel{\text{def}}{=} \{X_1, \ldots, X_m\}$ and outputs $\mathcal{Y} \stackrel{\text{def}}{=} \{Y_1, \ldots, Y_n\}$. To some extent, markers are like oid's, but unlike the latter, they can also be used in queries. It is not required that

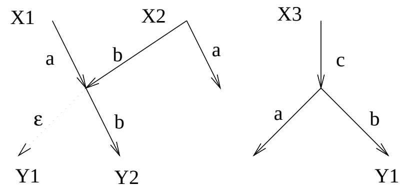


Figure 11: A database with inputs $\mathcal{X} = \{X_1, X_2, X_3\}$ and outputs $\mathcal{Y} = \{Y_1, Y_2\}$.

$\mathcal{X}$ and $\mathcal{Y}$ be disjoint: in fact there are cases when we take $\mathcal{X} = \mathcal{Y}$. We assume to have an infinite set of available markers, denoted $Marker$.

Recall that every semistructured database DB is equivalent (bisimilar) to its accessible part. For rooted graphs accessible meant "accessible from DB's root". When G is a graph with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$ its accessible part is that accessible from one of its inputs $X_1, \ldots, X_m$. The odd consequence is that not every output node is accessible. We accommodate this by allowing output nodes to be missing. Thus, to be precise, we define:

Definition 5.1 *Given two sets of markers $\mathcal{X} = \{X_1, \ldots, X_m\}$ and $\mathcal{Y} = \{Y_1, \ldots, Y_n\}$, a labeled graph with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$ is a graph whose edges are labeled with $Label \cup \{\varepsilon\}$, and having m distinguished nodes associated to the m input markers, and $n_0 \leq n$ distinguished leaves (i.e. nodes without outgoing edges) associated to some subset of the output markers $\mathcal{Y}$.*

Bisimulation carries over mutatis mutandis to graphs with inputs and outputs (see Appendix A). As before, a graph G with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$ is bisimilar to its accessible part, G^{acc}, which is also a graph with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$. Also we will relax Definition 5.1 and allow several leaves to be labeled with the same output marker $Y_i \in \mathcal{Y}$: such a graph is always bisimilar to one conforming to the definition. To see that it suffices to add n fresh nodes $v_1, \ldots, v_n$, label these and only these with the output markers $Y_1, \ldots, Y_n$ respectively, and add an ε edges from u to v_i whenever u was labeled Y_i in the original graph, for $i = 1, n$.

We denote with $Tree^{\mathcal{X}}_{\mathcal{Y}}$ the set of graphs with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$. The definition implies that $Tree^{X}_{\mathcal{Y}} \subseteq Tree^{X}_{\mathcal{Y}'}$ whenever $\mathcal{Y} \subseteq \mathcal{Y}'$

To recover the previously defined semi-structured databases, we assume a distinguished marker Δ to be given. Then a semi-structured databases as in Subsection 2.1 is a graph with inputs $\mathcal{X} = \{\Delta\}$ and outputs $\mathcal{Y} = \{\}$. That is $Tree = Tree^{\{\Delta\}}_{\emptyset}$.

Syntax We extend the syntax for trees to that of trees with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$. $T^{\mathcal{X}}_{\mathcal{Y}}$ denotes a tree with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$, and $T_{\mathcal{Y}}$ one with input $\{\Delta\}$ and outputs $\mathcal{Y}$:

$$\begin{array}{rcl} T^{X}_{\mathcal{Y}} & ::= & (X_1 := T_{\mathcal{Y}}; \ldots; X_m := T_{\mathcal{Y}}) \\ T_{\mathcal{Y}} & ::= & \{\} \mid \{Label \Rightarrow T_{\mathcal{Y}}\} \mid T_{\mathcal{Y}} \cup T_{\mathcal{Y}} \mid Y_j \ (j = 1, n) \end{array}$$

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[5] $(X_1 := \{a \Rightarrow (Y_1 \cup \{b \Rightarrow Y_2\})\}; X_2 := \{b \Rightarrow \{Y_1 \cup \{b \Rightarrow Y_2\}\}, a\}; X_3 := \{c \Rightarrow \{a, b \Rightarrow Y_1\}\})$.

5.2 Graph Operations

We revise here the graph constructors introduced in [BDHS96a, BDHS96b] for graphs with inputs and outputs. They allow us to construct more complex graphs from simpler ones.

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

Inputs and Outputs Whenever Y is a marker and $Y \in \mathcal{Y}$, the graph Y is in $Tree_{\mathcal{Y}}^{\{\Delta\}}$ and has a single node which is the designated input Δ and at the same time the output Y. Next, when $t_1, \ldots, t_m \in Tree_{\mathcal{Y}}^{\{\Delta\}}$ and $\mathcal{X} = \{X_1, \ldots, X_m\}$, then $(X_1 := t_1; \ldots; X_m := t_m)$ is a graph in $Tree_{\mathcal{Y}}^{\mathcal{X}}$ obtained as the disjoint union of $t_1, \ldots, t_m$, with their former roots labeled as inputs $X_1, \ldots, X_m$ respectively.

Empty graph, singleton graph, and union These three constructors are the same as before. Namely $\{\}$ denotes a graph with one node and no edges; that node is the root, and is labeled Δ. By definition $\{\}$ is in $Tree_{\mathcal{Y}}^{\{\Delta\}}$, for any set of markers $\mathcal{Y}$. Next, when $t \in Tree_{\mathcal{Y}}^{\{\Delta\}}$ and $a \in Label$, then the singleton tree is $\{a \Rightarrow t\} \in Tree_{\mathcal{Y}}^{\{\Delta\}}$. Given $t, t' \in Tree_{\mathcal{Y}}^{\{\Delta\}}$, we define $t \cup t' \in Tree_{\mathcal{Y}}^{\{\Delta\}}$ as in Figure 12 (a). Finally we extend the $\cup$ notation to trees with named inputs. Namely when $t, t' \in Tree_{\mathcal{Y}}^{\mathcal{X}}$, with $t = (X_1 := t_1; \ldots; X_m := t_m)$, $t' = (X_1 := t_1'; \ldots; X_m := t_m')$, then we define:

$$t \cup t' \stackrel{\text{def}}{=} (X_1 := t_1 \cup t_1'; \ldots; X_m := t_m \cup t_m')$$

Graph append Given two graphs $t \in Tree_{\mathcal{Y}}^{\mathcal{X}}$ and $t' \in Tree_{\mathcal{Z}}^{\mathcal{Y}}$, we define $t +\!\!+ t'$ to be the graph obtained by taking the disjoint union of their nodes and edges, and then adding n new ε edges, $Output_{Y_i}(t) \xrightarrow{\varepsilon} Input_{Y_i}(t')$, for every marker $Y_i \in \mathcal{Y}$ for which $Output_{Y_i}(t)$ exists (recall that not every output marker must have an associated output node); see Figure 12 (b). Modulo bisimulation, this is the same as substituting every output node in t with the corresponding input node (i.e. labeled with the same marker) in t'. Thus, append is a form of substitution. Namely when $E(Y_1, \ldots, Y_n)$ is some expressions with graph constructors and the output markers $Y_1, \ldots, Y_n$, then:

$$E +\!\!+ (Y_1 := t_1; \ldots; Y_n := t_n) = E[t_1/X_1, \ldots, t_n/X_n]$$

Identity graph Given a set of markers $\mathcal{X}$ we denote with $Id_{\mathcal{X}}$ the *identity graph*, $Id_{\mathcal{X}} \in Tree_{\mathcal{X}}^{\mathcal{X}}$, defined as:

$$Id_{\mathcal{X}} \stackrel{\text{def}}{=} (X_1 := X_1; \ldots; X_m := X_m)$$

This is the identity for append:

$$\begin{aligned} t +\!\!+ Id_{\mathcal{X}} &= t \\ Id_{\mathcal{X}} +\!\!+ t' &= t' \end{aligned}$$

Graph identity can be expressed in terms of the input/output constructs. But this requires naming the markers in X explicitly. When writing queries on a database fragment DB_α of a distributed database DB, we will use the notation $Id_{Inputs(DB_\alpha)}$ to denote the identity on DB_α's input markers.

Concatenation Given $t \in Tree_{\mathcal{Y}}^{\mathcal{X}}, t' \in Tree_{\mathcal{Y}}^{\mathcal{X}'}$ with $\mathcal{X} \cap \mathcal{X}' = \emptyset$, we define their *concatenation* $(t; t') \in Tree_{\mathcal{Y}}^{\mathcal{X} \cup \mathcal{X}'}$ to be the graph whose vertices and edges are the disjoint union of those in t, t', and in which we collapse similarly labeled output nodes, see Figure 12 (c) (to avoid clutter, we do not show the collapse of the output nodes). Note again the asymmetry between input and output nodes. Output nodes are leaves, so collapsing them or keeping them distinct makes no difference w.r.t. bisimulation. However if we were to collapse input nodes we would introduce new paths which were neither in t nor in t'. Hence we require the sets of input markers to be disjoint.

The null graph This is the graph with no edges and no vertices, and is denoted $()$. It is the unique (up to bisimulation) graph in $Tree_{\mathcal{Y}}^{\emptyset}$, for any $\mathcal{Y}$. It is the identity for concatenation:

$$((); t) = (t; ()) = t$$

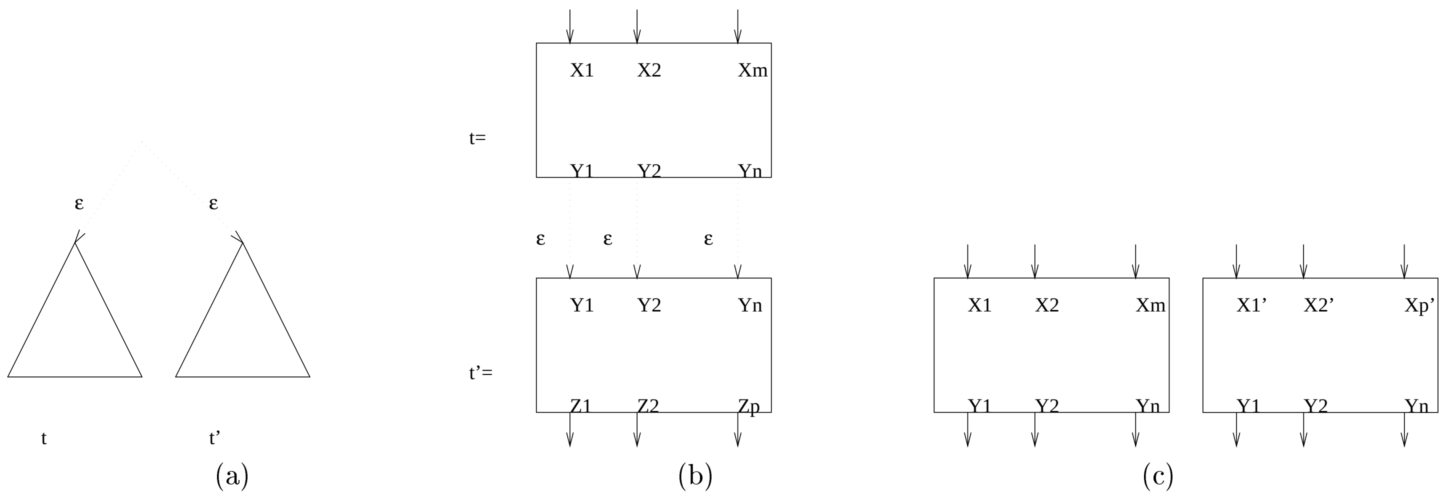


Figure 12: Illustration of $t \cup t'$, $t +\!\!+ t'$, and $(t; t')$.

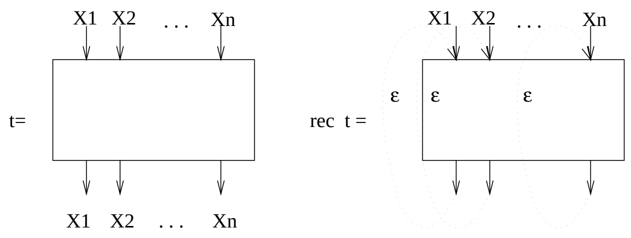


Figure 13: Illustration of $\mathsf{rec}\ t$ where t has inputs and outputs $\{X_1, \ldots, X_n\}$.

Recursion Given a graph $t \in Tree^{\mathcal{X}}_{\mathcal{X}}$, $\mathsf{rec}_{\mathcal{X}}\ t$ denotes a graph in $Tree^{\mathcal{X}}_{\emptyset}$ obtained by adding an ε-edge from every output X_i to the input X_i, $i = 1, m$ (Figure 13). For example $\mathsf{rec}_{\{X\}}\ (X := \{a \Rightarrow X\})$ defines a loop labeled a: its root is X. Furthermore, $X +\!\!+ \mathsf{rec}_{\{X\}}\ (X := \{a \Rightarrow X\})$ denotes the rooted labeled graph consisting of one loop a, see Figure 14. Notice that $\mathsf{rec}_{\mathcal{X}}\ t$ is similar to the infinite $t +\!\!+ t +\!\!+ t +\!\!+ \ldots$

Given these constructors we note that any graph can be expressed (not necessarily uniquely) in a **canonical form** as:

$$X_1 +\!\!+ \mathsf{rec}_{\mathcal{X}}\ (X_1 := t_1; \ldots, X_m := t_m) \tag{3}$$

for some set $\mathcal{X} = \{X_1, \ldots, X_m\}$, where each of t_i, $i = 1, m$ is cycle free.

5.3 Representing Distributed Databases

We show now how the above constructors allow us to represent formally distributed databases. 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:

$$DB = X_1 +\!\!+ \mathsf{rec}_{\mathcal{X}}\ (DB_1; \ldots; DB_m)$$

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.

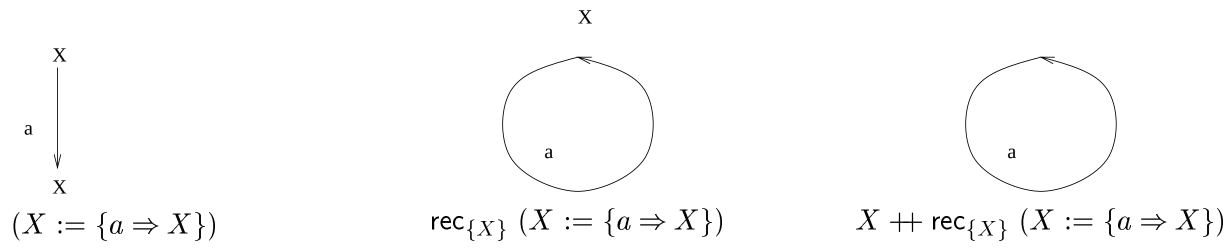

X
a
X
$(X := \{a \Rightarrow X\})$
X
a
$\mathsf{rec}_{\{X\}}\ (X := \{a \Rightarrow X\})$
a
$X \mathbin{+\!\!+} \mathsf{rec}_{\{X\}}\ (X := \{a \Rightarrow X\})$

Figure 14: Example of the use of rec and ++ constructs.

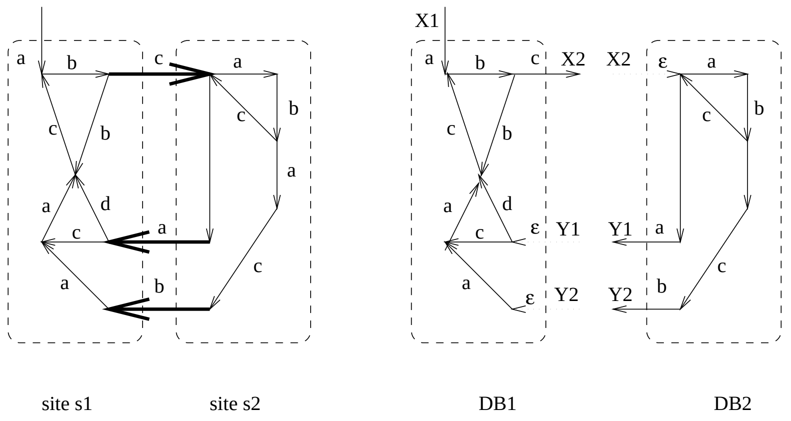


Figure 15: Representation of a distributed database.

We illustrate with the example in Figure 15, where DB 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 two fragments with these minor cosmetic changes: here $DB_1 \in Tree_{\mathcal{X}}^{\mathcal{X}_1}$ and $DB_2 \in Tree_{\mathcal{X}}^{\mathcal{X}_2}$, 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 $DB \in Tree$ is represented as:

$$DB = X_1 \mathbin{+\!\!+} (\mathsf{rec}_{\mathcal{X}}\ (DB_1; DB_2))$$

where $(DB_1; DB_2)$ is the concatenation of DB_1 and DB_2. The $\mathsf{rec}_{\mathcal{X}} \ldots$ construct redraws the cross links, while $X_1 \mathbin{+\!\!+} (\ldots)$ selects only the input labeled X_1 as the unique input of the database.

5.4 A Calculus for Graphs

We describe next a calculus, $\mathcal{C}$, for computing with graphs. $\mathcal{C}$ is a fragment of UnCAL described in [BDHS96a]: it is join-free and positive. $\mathcal{C}$ consists of all graph constructors described in the previous subsection, plus an iterator and a conditional. The conditional is of the form if P then E_1 else E_2, where P is a predicate and E_1, E_2 are expressions in $\mathcal{C}$. Predicates are boolean combinations of either unary predicates on labels $(P(l))$, or label equality $(l = l')$, where l, l' are label variables or label constants. As for selection queries, we denote the query's input with DB.

The iterator is more complex and we will describe it next. When applied to graphs with named input and output markers, the iterator returns a graph with new markers: we start by introducing a marker constructor. Recall that $Marker$ is the set of all markers. Given $X, Y \in Marker$, we denote with $X \cdot Y$ a new marker, distinct from X and Y. We require $\cdot$ to be injective on

$Marker - \{\Delta\}$, i.e. $X \cdot Y = X' \cdot Y'$ implies $X = X'$ and $Y = Y'$; we also require Δ to be an identity, i.e. $X \cdot \Delta = \Delta \cdot X = X$. If one thinks of markers as oid's, then $X \cdot Y$ is oid creation.

The iterator construct is $iter_{\mathcal{S}}(l.B)(E)$. Here B is the body of the iterator, and can be any expressions which may use the label variable l, but not the input DB: we call B a *constant expression*. B returns a graph in $Tree^{\mathcal{S}}_{\mathcal{S}}$: note that it has the same set of input and output markers, $\mathcal{S} = \{S_1, \ldots, S_k\}$. Assume first that $E \in Tree$, i.e. has no input or output markers. One way to describe the meaning of $iter_{\mathcal{S}}(l.B)(E)$ is as a parallel substitution. First compute E to obtain a graph t. Next, for every edge e in t, say e is $u \xrightarrow{a} v$, compute $B[a/l]$: call this value b_e, and note that it has inputs/outputs $\mathcal{S}$. Finally "merge" together all graphs b_e according to the connection between edges in the original graph t. That is if $e_1, e_2, \ldots$ all have the same source node u, then join the inputs of $b_{e_1}, b_{e_2}, \ldots$ with the same markers, and if these edges share the same output, then join the corresponding outputs of $b_{e_1}, b_{e_2}, \ldots$ This is tantamount to (1) replacing every node u in t with k fresh nodes corresponding to the markers in $\mathcal{S}$, call them $(S_1, u), \ldots, (S_k, u)$, and (2) for every edge $e : u \to v$, connecting $(S_1, u), \ldots, (S_k, u)$ to the k inputs of b_e with ε edges, and the k outputs of b_e to $(S_1, v), \ldots, (S_k, v)$, also with ε edges. Finally we note that the resulting graph will have inputs $\mathcal{S}$ (those corresponding to the original root of t) and no outputs.

An alternative way to describe $iter(l.B)(E)$ is by recursion on t (the value of E), as follows:

$$\begin{aligned}
iter_{\mathcal{S}}(l.B)(\{\}) &\stackrel{\text{def}}{=} \{\} \\
iter_{\mathcal{S}}(l.B)(\{a \Rightarrow t\}) &\stackrel{\text{def}}{=} B[a/l] +\!\!+\; iter_{\mathcal{S}}(l.B)(t) \\
iter_{\mathcal{S}}(l.B)(t_1 \cup t_2) &\stackrel{\text{def}}{=} iter_{\mathcal{S}}(l.B)(t_1) \cup iter_{\mathcal{S}}(l.B)(t_2)
\end{aligned}$$

The last $\cup$ operator is applied to two graphs in $Tree^{\mathcal{S}}$: recall from Subsection 5.2 that this just means that union is taken component-wise.

This definition makes perfect sense when t is a tree, and it is not hard to see that it produces the same result (up to bisimulation) as the the parallel substitution described above (a formal proof is given in [BDHS96b]). When t is a graph with cycles then the above definition could be read as either incomplete or as non-terminating, since it chases loops forever by recursively calling $iter_{\mathcal{S}}(l.B)$. An important result in [BDHS96a] is the observation that this recursive definition still makes perfect sense on graphs with cycles, if one interprets it as memorizing the visited nodes, and avoiding entering infinite loops, much in the same spirit as the function *visit* of Subsection 3.2. Moreover, this interpretation always coincides with that given by the parallel substitution.

We describe now the meaning of $iter_{\mathcal{S}}(l.B)(E)$ for the case when E denotes a graph in $Tree^{\mathcal{X}}_{\mathcal{Y}}$. In this case the result will be a graph in $Tree^{\mathcal{S}\cdot\mathcal{X}}_{\mathcal{S}\cdot\mathcal{Y}}$. For the parallel substitution, the intuition is the following. Recall that we replace each node u with k copies, $(S_1, u), \ldots, (S_k, u)$. If the node u was in input node labeled X, then these copies will be the input nodes labeled $S_1 \cdot X, \ldots, S_k \cdot X$. Similarly if u was an output node labeled Y. For the recursive definition of *iter*, it suffices to add two clauses dealing with input and output markers:

$$\begin{aligned}
iter_{\mathcal{S}}(l.B)(Y) &\stackrel{\text{def}}{=} (S_1 := S_1 \cdot Y; \ldots; S_k := S_k \cdot Y) \\
iter_{\mathcal{S}}(l.B)(X_1 := t_1; \ldots; X_m := t_m) &\stackrel{\text{def}}{=} (iter_{\mathcal{S}}(l.B)(t_1) \cdot X_1; \ldots iter_{\mathcal{S}}(l.B)(t_m) \cdot X_m)
\end{aligned}$$

Here we used a new notation: $t \cdot X$ denotes the same data graph as t, but with each input S is relabeled as input $S \cdot X$.

Definition 5.2 *We define the calculus $\mathcal{C}$ to consist of the operators in Table 1. Expressions may have one free label variable (denoted l), and may refer to DB. A* query *is any closed expression in $\mathcal{C}$. A* constant expression *is one in which DB doesn't appear.*

Operator	Name
DB	input database
$\{\}$	empty graph
$\{a \Rightarrow Q\}$	singleton graph
$Q \cup Q'$	union
Y	output marker
$(X_1 := Q_1; \ldots X_m := Q_m)$	input markers
$Q \mathbin{+\!\!+} Q'$	append
$Id_{\mathcal{X}}$	identity
$(Q; Q')$	concatenation
$()$	null graph
$\mathsf{rec}_{\mathcal{X}}\ Q$	recursion
$iter_{\mathcal{S}}(l.B)(Q)$	iteration
$\mathsf{if}\ P(l)\ \mathsf{then}\ Q\ \mathsf{else}\ Q'$	conditional

Table 1: Operators in $\mathcal{C}$. Here Q, Q' are subexpressions, and B is a constant subexpression.

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 $Pub(l)$ which decides whether the label l 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 as:

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

The combined function $\varphi(t) \stackrel{\text{def}}{=} (S_1 := \varphi_1(t); S_2 := \varphi_2(t))$ can be expressed as a single *iter* construct. Namely define $\mathcal{S} \stackrel{\text{def}}{=} \{S_1, S_2\}$ and the body:

$$\begin{aligned} B(l) \stackrel{\text{def}}{=} (\ & S_1 := \mathsf{if}\ l = \text{“}CSDept\text{”}\ \mathsf{then}\ \{l \Rightarrow S_2\}\ \mathsf{else}\ \{l \Rightarrow S_1\}; \\ & S_2 := \mathsf{if}\ Pub(l)\ \mathsf{then}\ \{\text{“}TR\text{”} \Rightarrow S_2\}\ \mathsf{else}\ \{l \Rightarrow S_2\}) \end{aligned}$$

Then $\varphi(t) = iter_{\mathcal{S}}(l.B)(t)$ and the query can be expressed as $Q(DB) = S_1 \mathbin{+\!\!+} iter_{\mathcal{S}}(l.B)(DB)$. For the database DB in Figure 16 (a), $iter_{\mathcal{S}}(l.B)$ is shown in (c). The simplified form (under bisimulation) of $S_1 \mathbin{+\!\!+} iter_{\mathcal{S}}(l.B)(DB)$ is shown in (b). □

This example illustrates two ideas. First that $\mathcal{C}$ can express more complex restructuring queries than those expressed by selection queries: the query in this example cannot be expressed using select. Second the evaluation of *iter* can be done locally at each edge, independent of the other edges: we will exploit this when we show that every query in $\mathcal{C}$ can be decomposed.

In the remainder of this section we show that all extended regular queries are expressible in $\mathcal{C}$.

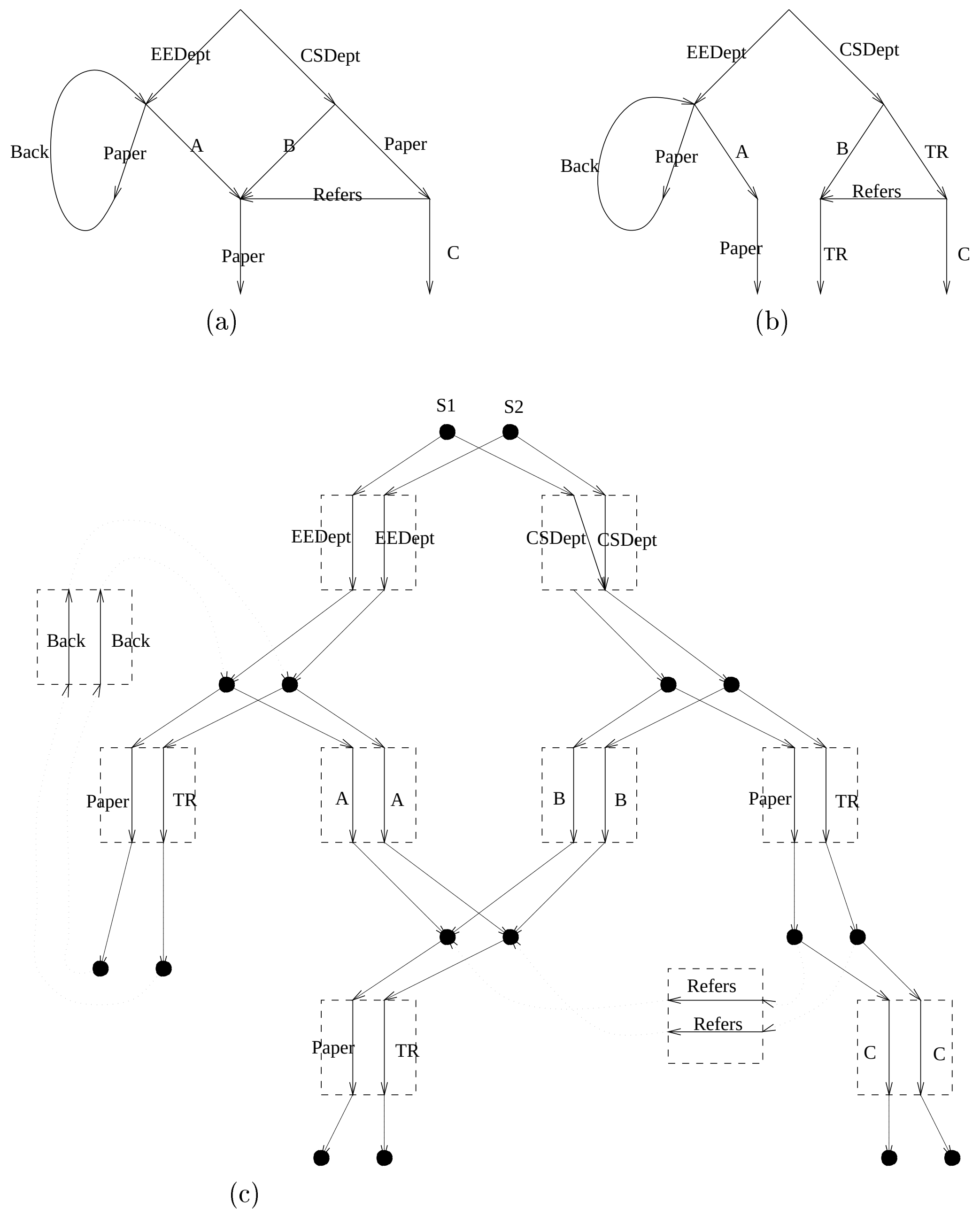


Figure 16: Illustration for Example 5.5. A Database (a), the result of $iter_{\mathcal{S}}(l.B)(DB)$ (c), and the simplified answer of $Q(DB) = S_1 +\!\!+\; iter(l.B)(DB)$ (b).

Theorem 5.4 *Every extended regular $Q(DB)$ query can be expressed as $P +\!\!+\, iter_{\mathcal{S}}(l.B)(DB)$, for some set of markers $\mathcal{S}$, where P is a constant expression.*

Proof: Let $Q(x,t)$ be an extended regular query, as in Definition 4.2. We prove by induction on Q that there exists n mutually recursive functions $\varphi_1(t), \ldots, \varphi_n(t)$ and some expression P involving only constructors such that $Q(x,t) = P(x, \varphi_1(t), \ldots, \varphi_n(t))$. Then the theorem follows from the fact that $Q(DB) = P(\varphi_1(DB), \ldots, \varphi_n(DB)) = P(S_1, \ldots, S_n) +\!\!+ (S_1 := \varphi_1(DB); \ldots; S_n := \varphi_n(DB))$: the latter is a *iter* expression. Consider Case (1) of Definition 4.2: $Q(t) = t$. Here we take $n = 1$, and $\varphi_1(t)$ to be the identity function $\varphi_1(t) = t$ (see Example 5.5 below). In Case (2), $Q(t) = \{\}$, we take $P = \{\}$. Consider Case (3), when $Q(x,t) = \{a \Rightarrow Q_1(x,t)\}$. Apply induction hypothesis to obtain $Q_1(x,t) = P_1(x, \varphi_1(t), \ldots, \varphi_n(t))$. Then Q can be expressed as $\{a \Rightarrow P_1(x, \varphi_1(t), \ldots, \varphi_n(t))\}$. Case (4) is similar. Now consider case (6):

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

Apply induction hypothesis to get $Q_1(x_1,t_1) = P_1(x_1, \varphi_1(t_1), \ldots, \varphi_n(t_1))$, and also let $\psi_1(t), \ldots, \psi_m(t)$ be m mutually recursive functions implementing the regular expression R, i.e. $\psi_1(t) = \mathsf{select}\ t'\ \mathsf{where}\ R \Rightarrow t'\ \mathsf{in}\ t$, with the assumption $\psi_m(t) \equiv t$. Such functions can be constructed easily from R's automaton, as we illustrate in Example 5.5 below. Next replace ψ_m with ψ'_m defined as:

$$\begin{aligned}
\psi'_m(\{\}) &= \{\} \\
\psi'_m(\{x_1 \Rightarrow t_1\}) &= \mathsf{if}\ P(x_1)\ \mathsf{then}\ P_1(x_1, \varphi_1(t_1), \ldots, \varphi_n(t_1))\ \mathsf{else}\ \{\} \\
\psi'_m(t \cup t') &= \psi'_m(t) \cup \psi'_m(t')
\end{aligned}$$

Now we have $m+n$ mutually recursive functions, $\psi_1(t), \ldots, \psi_{m-1}(t), \psi'_m(t), \varphi_1(t), \ldots, \varphi_n(t)$. Then $Q(x,t) = \psi_1(t)$. Case (5) is left to the reader. □

Example 5.5 Consider the regular query

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

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

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

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

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

Notice that φ_3, which corresponds to the terminal state, is the identity function. Next, the function $\varphi(t) \stackrel{\text{def}}{=} (S_1 := \varphi_1(t); S_2 := \varphi_2(t); S_3 := \varphi_3(t))$ can be expressed as a *iter* construct $\varphi(t) = iter_{\mathcal{S}}(l.B)(t)$, where $\mathcal{S} = \{S_1, S_2, S_3\}$ and the body is:

$$\begin{aligned}
B(l) \stackrel{\text{def}}{=} (\ & S_1 := \mathsf{if}\ \mathit{CSDept}(l)\ \mathsf{then}\ S_1 \cup S_2\ \mathsf{else}\ S_1; \\
& S_2 := \mathsf{if}\ \mathit{Paper}(l)\ \mathsf{then}\ S_2 \cup S_3\ \mathsf{else\ if}\ \mathit{not}(\mathit{Dept}(l))\ \mathsf{then}\ S_2\ \mathsf{else}\ \{\} \\
& S_3 := \{l \Rightarrow S_3\})
\end{aligned}$$

Thus, the query Q is equivalent to $S_1 +\!\!+\, iter_{\mathcal{S}}(l.E)(DB)$. □

Example 5.6 Consider now a more complex extended regular query:

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

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

$$\begin{aligned}\varphi_1(\{x \Rightarrow t\}) &= \{x \Rightarrow \varphi_2(t) \cup \varphi_3(t)\}\\ \varphi_2(\{l \Rightarrow t'\}) &= \textsf{if } l = “B'' \textsf{ then } \varphi_3(t') \textsf{ else } \{\}\end{aligned}$$

and $\varphi_3(t) = t$ is the identity function (we omitted the clauses $\varphi_i(\{\}) = \{\}$ and $\varphi_i(t \cup t') = \varphi_i(t) \cup \varphi_i(t')$). Hence $Q(DB) = S_1 +\!\!+\ iter_{\mathcal{S}}(l.B)(DB)$, where:

$$\begin{aligned}B(l) = (\ &S_1 := \{l \Rightarrow S_2 \cup S_3\};\\ &S_2 := \textsf{if } l = “B'' \textsf{ then } S_3 \textsf{ else } \{\};\\ &S_3 := \{l \Rightarrow S_3\})\end{aligned}$$

□

5.5 Decomposed Queries and Distributed Evaluation

Our key technique for query decomposition is to transform queries into *decomposed* queries.

Definition 5.7 *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', $Q(t; t') = (Q(t); Q(t'))$.*

If $Q(t) = iter_{\mathcal{S}}(l.B)(t)$, then Q is decomposed: condition 1 follows from [BDHS96a], while condition 2 is easy to check. If Q_1, Q_2 are decomposed queries then so is their concatenation, $Q(t) \overset{\text{def}}{=} (Q_1(t), Q_2(t))$, if this operation makes sense. Finally, the identity function, $Q(t) \overset{\text{def}}{=} t$, is a decomposed query. These are the only kind of decomposed queries we will use: they can be further simplified because one can show that a query obtained by concatenating *iter* expressions is in turn a single *iter* expression, but we do not need that in the sequel.

Proposition 5.8 *If Q is decomposed then:*

$$\begin{aligned}&Q(\textsf{rec } (X_1 := t_1; \ldots X_k := t_k)) =\\ &\qquad \textsf{rec } (Q(X_1 := t_1); \ldots Q(X_k := t_k))\end{aligned}$$

Proof: We only give an informal argument. Since $\textsf{rec } t$ is the same as the infinite unfolding $t +\!\!+\ t +\!\!+ \ldots$, we have $Q(\textsf{rec } t) = Q(t +\!\!+\ t +\!\!+ \ldots) = Q(t) +\!\!+\ Q(t) +\!\!+ \ldots = \textsf{rec } Q(t)$. Next we apply item 2 of Definition 5.7. □

Our main interest in decomposed queries is that they can be efficiently evaluated on distributed databases (in the sense of Definition 1.1):

```
Algorithm : Distributed-Evaluation-C
Input :     A query of the form $P +\!\!+ Q(DB)$
                    with $P$ constant and $Q$ decomposed,
            A semistructured database $DB$ distributed on a number of sites:
                    $DB = X_1 +\!\!+ \mathsf{rec}_{\mathcal{X}}\ (DB_1; \ldots; DB_m)$
Output :    Evaluates $Q(DB)$
Method :
            Step 1  Send $Q$ to all servers $\alpha$, $\alpha = 1, m$.
            Step 2  At every site $\alpha$ compute $F_\alpha := Q(DB_\alpha)$
            Step 3  At every site $\alpha$ construct the accessibility graph from $F_\alpha$ (see text)
            Step 4  Every site $\alpha$ 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 $\alpha$, $\alpha = 1, m$.
            Step 6  Every site $\alpha$ computes $F_\alpha^{acc}$, the accessible part of $F_\alpha$.
            Step 7  Every site $\alpha$ sends $F_\alpha^{acc}$ to the client
                    which computes the final result $X_1 +\!\!+ \mathsf{rec}_{\mathcal{X}}\ (F_1^{acc}; \ldots; F_m^{acc})$.
```

Figure 17: Distributed evaluation.

Theorem 5.9 *Algorithm Distributed-Evaluation-C in Figure 17 efficiently computes a query of the form $P +\!\!+ Q(DB)$, where P is a constant expression and Q is decomposed, on a distributed database DB. Specifically:*

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.*

Proof: Recall that a distributed database DB can be represented as $DB = X_1 +\!\!+ \mathsf{rec}\ (DB_1; \ldots; DB_m)$, where $DB_1, \ldots, DB_m$ are the fragments of the distributed database. Then $Q(DB) = Q(X_1) +\!\!+ \mathsf{rec}\ (Q(DB_1); \ldots; Q(DB_m))$. Thus we start by evaluating Q independently on each fragment DB_α $\alpha = 1, m$. As in Algorithm *Distributed-Evaluation* , we perform some trimming before shipping the fragments to the client site, since after final assembly large pieces of the graph may be unaccessible. Namely for each fragment F_α we compute, in Step 4, the accessibility graph, telling which input markers are connected to which output markers. All these graphs are centralized at the client, which combines them with the accessibility graph of P (saying which output markers are accessible from P's root). The client computes all nodes accessible from P's root, and broadcasts them to the servers. These compute F_α^{acc}, the accessible part of F_α, and these fragments are sent to the client and assembled. $\square$

5.6 Query Decomposition

We prove here the main result of this section, that every query in $\mathcal{C}$ can be expressed as $P \mathbin{+\!\!+} Q^{dec}(DB)$, where Q^{dec} is decomposed. In combination with Theorem 5.9, this gives us an efficient evaluation algorithm for queries in $\mathcal{C}$ on distributed databases.

Theorem 5.10 *For every query Q in $\mathcal{C}$ there exists a decomposed query Q^{dec} and constant expression P such that $Q(DB) = P \mathbin{+\!\!+} Q^{dec}(DB)$.*
We call the process of finding P, Q^{dec} for a given Q "query decomposition".

Proof: We prove by induction on the structure of Q (see Table 1). Before that, we make some general remarks. First we never apply induction to the body of *iter* constructs: hence our subexpressions will not have free label variables, and we do not need to consider the case $\mathsf{if}\ P(l)\ \mathsf{then}\ Q\ \mathsf{else}\ Q'$. Second, recall that our original query Q is applied to some database DB without output markers (the original database is in $Tree_{\emptyset}^{\{\Delta\}}$). Hence one can see that in every subexpression $Q_1(DB) \mathbin{+\!\!+} Q_2(DB)$ either Q_1 is constant (i.e. does not depend on DB), or otherwise cannot have output markers[6], hence $Q_1(DB) \mathbin{+\!\!+} Q_2(DB) = Q_1(DB)$. Finally, we pre-process the entire query Q such as to rename the local markers $\mathcal{S}$ in every $iter_{\mathcal{S}}$ construct to be disjoint from all other markers used in other *iter* constructs. We these observations in mind, we proceed to the proof by induction.

1. $Q(DB) = DB$. Then take $P \stackrel{\text{def}}{=} Id_{Inputs(DB)}$ and $Q^{dec} \stackrel{\text{def}}{=} DB$.

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 $Q_1(DB) = P_1 \mathbin{+\!\!+} Q_1^{dec}(DB)$. 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 $Q_1(DB) = P_1 \mathbin{+\!\!+} Q_1^{dec}(DB)$ and $Q_2(DB) = P_2 \mathbin{+\!\!+} Q_2^{dec}(DB)$. In short, here we define $P \stackrel{\text{def}}{=} P_1 \cup P_2$ and $Q^{dec}(DB) \stackrel{\text{def}}{=} (Q_1^{dec}(DB); Q_2^{dec}(DB))$. But we must take some precautions to make sure that $(Q_1^{dec}(DB); Q_2^{dec}(DB))$ 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 *iter* constructs have disjoint sets of markers) reveals that $Q_1^{dec}(DB)$ can either (a1) only have private input markers, not shared by $Q_2^{dec}(DB)$, or (b1) be of the form $((Q_1^{dec})'(DB); DB)$. Similarly, $Q_2^{dec}(DB)$ can either (a2) have only private input markers, or (b2) be of the form $((Q_2^{dec})'(DB); DB)$. For the first three of the four combined cases, it is safe to define $Q^{dec}(DB) \stackrel{\text{def}}{=} (Q_1^{dec}(DB); Q_2^{dec}(DB))$. For the last case we notice that the part with common input markers is actually identical, and we define $Q^{dec}(DB) \stackrel{\text{def}}{=} ((Q_1^{dec})'(DB); (Q_2^{dec})'(DB); DB)$.

5. $Q(DB) = Q_1 \mathbin{+\!\!+} Q_2(DB)$. It suffices to consider the case when Q_1 is a constant expression. We apply induction hypothesis first to Q_2, $Q_2(DB) = P_2 \mathbin{+\!\!+} Q_2^{dec}(DB)$, then define $P \stackrel{\text{def}}{=} Q_1 \mathbin{+\!\!+} P_2$, $Q^{dec}(DB) \stackrel{\text{def}}{=} Q_2^{dec}(DB)$.

6. $Q(DB) = iter_{\mathcal{S}}(l.B)(Q_1(DB))$, where B is a constant query. We apply induction hypothesis and decompose $Q_1(DB)$ into $P_1 \mathbin{+\!\!+} Q_1^{dec}(DB)$. Then we define $P \stackrel{\text{def}}{=} iter_{\mathcal{S}}(l.B)(P_1)$ and $Q^{dec}(DB) = iter_{\mathcal{S}}(l.B)(Q_1^{dec}(DB))$.

[6]A mixture of the two may hold. We leave some details to the reader.

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. $Q(DB) = (Q_1(DB); Q_2(DB))$. Decompose the subqueries first: $Q_i(DB) = P_i ++ Q_i^{dec}(DB)$, $i = 1, 2$, then define $P \stackrel{\text{def}}{=} (P_1; P_2)$, $Q \stackrel{\text{def}}{=} (Q_1^{dec}; Q_2^{dec})$.

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

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

$\square$

Summarizing, we have:

Corollary 5.11 *Every query in $\mathcal{C}$ can be efficiently evaluated on distributed databases. In consequence, every extended regular query can be efficiently evaluated on distributed databases.*

6 Distributed Evaluation of Selection Queries

Since algorithm *Distributed-Evaluation-$\mathcal{C}$* applies to all queries in $\mathcal{C}$, it also applies to extended regular queries. However it does not work for more complex selection queries, like queries $Q2, Q3, Q4$ of Subsection 4.1, which cannot be expressed in $\mathcal{C}$. We will describe here a more complex algorithm for efficiently evaluation of arbitrary, join-free selection queries on distributed databases. This may seem surprisingly, since selection queries can exhibit a more complex behavior than extended regular queries or queries in $\mathcal{C}$: some difficulties were highlighted in Subsection 4.1. In this algorithm we use two new techniques: *partial evaluation* and *alternating graph accessibility*. To evaluate a selection query Q, we start by evaluating a different (but related) query Q_r: the new query is an extended regular 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* [Imm87, GHR95], which generalizes the graph accessibility problem.

We illustrate here the ideas behind both techniques. To illustrate partial evaluation, consider the following selection query:

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

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. It also "shuffles" x, y, t_1, t_2, by grouping y with t_1 and x with t_2. With partial evaluation we evaluate first the following query:

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

Here $P \overset{\text{def}}{=} Q_r(DB)$ is the partial result, and Q_r is an extended regular query, which we know how to evaluate distributively. However P is different from the actual result $Q(DB)$, 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:

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

Note that the size of P is less than or equal to that of $Q(DB)$ (modulo some constant), except for the case when $n_1 = 0$ or $n_2 = 0$, when P can be arbitrarily large while $Q(DB)$ is empty: for that reason we have to avoid sending the t_1's to the client when $n_2 = 0$, and vice versa. This is the purpose of the alternating graph accessibility.

To summerize, $Q_r(DB)$ ignores the select clauses of Q, and constructs a graph containing all variable bindings, in the order which which they are introduced in the where clause: for the general case, it will have a more complex structure than in our example.

To illustrate best the alternating graph accessibility, consider another selection query:

$$\begin{aligned} Q(DB) = {} & \mathsf{select}\ \{\text{``}A\text{''} \Rightarrow \{\text{``}B\text{''} \Rightarrow t_1, \text{``}C\text{''} \Rightarrow t_2, \text{``}D\text{''} \Rightarrow t_3\}\} \\ & \mathsf{where}\ * \Rightarrow \text{``}B\text{''} \Rightarrow t_1\ \mathsf{in}\ DB \\ & \qquad * \Rightarrow \text{``}C\text{''} \Rightarrow t_2\ \mathsf{in}\ DB \\ & \qquad * \Rightarrow \text{``}D\text{''} \Rightarrow t_3\ \mathsf{in}\ DB \end{aligned}$$

which we replace with the following extended regular query:

$$\begin{aligned} Q_r = \{\text{``}A\text{''} \Rightarrow {} & (\mathsf{select}\ \{\text{``}B\text{''} \Rightarrow t_1\}\ \mathsf{where}\ * \Rightarrow \text{``}B\text{''} \Rightarrow t_1\ \mathsf{in}\ DB) \cup \\ & (\mathsf{select}\ \{\text{``}C\text{''} \Rightarrow t_2\}\ \mathsf{where}\ * \Rightarrow \text{``}C\text{''} \Rightarrow t_2\ \mathsf{in}\ DB) \cup \\ & (\mathsf{select}\ \{\text{``}D\text{''} \Rightarrow t_3\}\ \mathsf{where}\ * \Rightarrow \text{``}D\text{''} \Rightarrow t_3\ \mathsf{in}\ DB)\} \end{aligned}$$

Suppose one of the servers storing a distributed database holds t_1, which is a candidate match of the $* \Rightarrow \text{``}B\text{''} \Rightarrow t_1$ in DB condition. Should t_1 be sent to the client ? Of course, the root of t_1 must be accessible from the original database root, as before: this is the accessibility problem. But in addition we also have to check that there exists matches for *both* t_2 and t_3. That is, we have to find not *one*, but *three* paths from t_1's root: one to DB's root, one to some t_2, and one to some t_3. Formalized properly, this is precisely the AGAP problem.

To summarize, the alternating graph accessibility helps us keep track of the relationship between the subblocks of a select − where block. If one of the subblocks is empty, then all bindings done for the other subblocks have to be dropped. In the example above, there are three subblocks in the unique select − where block of $Q(DB)$. If one of them, say the middle one, is empty, then all bindings for t_1 and t_3 have to be discarded.

We describe the details in the remainder of this section.

6.1 Translating Select Queries into Extended Regular Queries

The idea behind the regular query Q_r associated to some selection query Q is simple. Q_r essentially collects all bindings of all variables occurring in Q, grouped conveniently to make Q_r an extended regular query; hence Q_r can be evaluated distributively to obtain the partial result $P \overset{\text{def}}{=} Q_r(DB)$. In a second step, we restructure P such as to obtain $Q(DB)$. This restructuring doesn't need to be a regular query, since it is computed locally, at the client. It turns out that the restructuring is another selection 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 selection 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 selection query Q, we define its pattern tree PT by induction on Q's structure. PT only depends on the where clauses in Q and its subqueries, not on the select parts; subqueries, the corresponding pattern tree will be in fact a pattern forest. PT will have nodes labeled with tree variables occurring in Q, and edges labeled either with label variables or regular expressions (both occurring in Q). Let $Q'(t_1, \ldots, t_k)$ be a sub-selection query of Q, which uses the free tree variables $t_1, \ldots, t_k$ in its where clause. That is Q' is:

select E

where P_1 in $s_1, \ldots, P_n$ in s_n

where each of $t_1, \ldots, t_k$ occurs at least once among $s_1, \ldots, s_n$. The roots of the pattern forest correspond to $t_1, \ldots, t_k$, and are labeled with $t_1, \ldots, t_k$ respectively. The pattern forest will have one node for every tree variable in the where clause of Q'. For each generator $P_i \Rightarrow t$ in s_i, the pattern forest has an edge $s_i \to t$ labeled P_i; for each generator $x_i \Rightarrow t$ in s_i the pattern forest has one edge $s_i \to t$ labeled x_i. For every select subquery Q'' occurring in E, we construct its pattern forest PT'': for each of the roots t of PT'' we add an ε-edge from the node t in PT' to the root t in PT''.

Thus, for the initial query, PT will have DB at its root, and may have ε edges connecting identically labeled nodes. If we delete all the ε edges, PT breaks into a number of connected components: we call each such component a *subblock*. A select statement with k free tree variables $t_1, \ldots, t_k$ in its generators, will correspond precisely to k subblocks: we call *block* the union of all subblocks corresponding to a select statement.

Example 6.1 Consider the query:

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

where R, R_1, R_2 are regular path expressions. It's associated pattern tree is shown in Figure 18. There are three subblocks: 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:

$$\begin{aligned} Q = \text{select } \{ \text{“}A'' \Rightarrow (\text{ select } \{ \text{“}AA'' \Rightarrow (& \text{ select } \{\text{“}AAA'' \Rightarrow t\} \\ & \text{ where } R_{11} \Rightarrow t_{11} \text{ in } t_1, R_{12} \Rightarrow t_{12} \text{ in } t_1 \end{aligned}$$

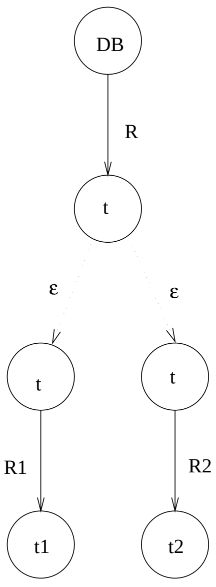


Figure 18: Pattern Tree for Example 6.1

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad\qquad\qquad R_{21} \Rightarrow t_{21} \text{ in } t_2, R_{22} \Rightarrow t_{22} \text{ in } t_2)\}\\
&\qquad\qquad \text{where } R_1 \Rightarrow t_1 \text{ in } t, R_2 \Rightarrow t_2 \text{ in } t),\\
&\quad \text{“}B'' \Rightarrow (\text{ select } \{\ \text{“}BB'' \Rightarrow (\text{ select } \{\text{“}BBB'' \Rightarrow t\}\\
&\qquad\qquad\qquad\qquad\qquad \text{where } R_{31} \Rightarrow t_{31} \text{ in } t_3, R_{32} \Rightarrow t_{32} \text{ in } t_3\\
&\qquad\qquad\qquad\qquad\qquad\qquad R_{41} \Rightarrow t_{41} \text{ in } t_4, R_{42} \Rightarrow t_{42} \text{ in } t_4)\}\\
&\qquad\qquad \text{where } R_3 \Rightarrow t_3 \text{ in } t, R_4 \Rightarrow t_4 \text{ in } t)\}\\
&\text{where } R \Rightarrow t \text{ in } DB
\end{aligned}
$$

The pattern tree is shown in Figure 19. There are five select blocks, three consisting of one subblock, two of two subblocks. Hence the pattern tree has seven subblocks. □

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: $s \overset{\text{def}}{=} \{\text{“}Match\text{”} \Rightarrow m_1, \ldots, \text{“}Match\text{”} \Rightarrow m_k\}$, 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

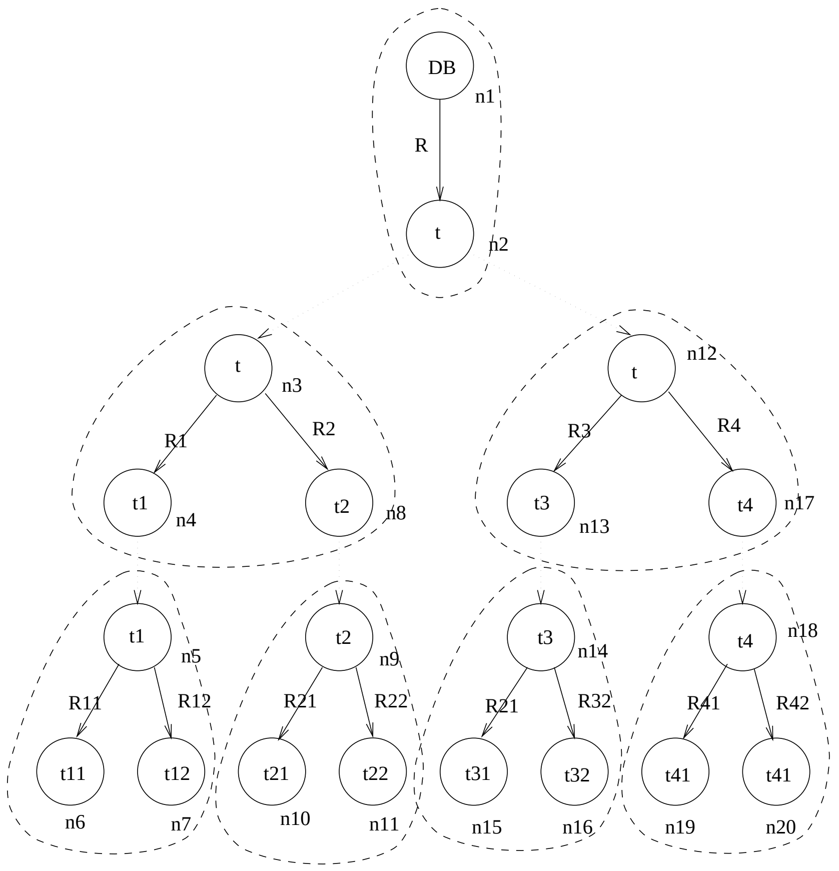


Figure 19: Pattern Tree for Example 6.2

the form: $m_i = \{``x'' \Rightarrow \{x\}, ``t'' \Rightarrow t, ``1'' \Rightarrow s_{i1}, ``2'' \Rightarrow s_{i2}, \ldots, ``l'' \Rightarrow s_{il}\}$, where the trees $s_{i1}, \ldots, s_{il}$ are match-set nodes for $t'_1, \ldots, t'_l$ respectively. Here $``1''$, $``2''$, $\ldots$, $``l''$ are just labels representing the successors number. Of course, if there is a regular expression R instead of a label variable x, then $m_i = \{``t'' \Rightarrow t, ``1'' \Rightarrow s_{i1}, ``2'' \Rightarrow s_{i2}, \ldots, ``l'' \Rightarrow s_{il}\}$.

Partial Query No matter how complex the selection query Q is, the partial result P can obtained with an extended regular 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:

$$M_n(x,t) \stackrel{\text{def}}{=} \{``x'' \Rightarrow \{x\}, ``t'' \Rightarrow t, ``1'' \Rightarrow S_{n_1}(t), \ldots, ``k'' \Rightarrow S_{n_k}(t)\}$$

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 $\varepsilon$ 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 $S_{n_i}(t)$ or $S_{n_i}(x,t)$ constructs the match-set node of that successor node. When the edge $n \to n_i$ is a ε edge, then:

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

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

$$\begin{aligned} S_{n_i}(t) = {} & \mathsf{select}\ \{\,\textit{``Match''} \Rightarrow M_{n_i}(t_i)\} \\ & \mathsf{where}\ R \Rightarrow t_i\ \mathsf{in}\ t \end{aligned}$$

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

$$\begin{aligned} S_{n_i}(t) = {} & \mathsf{select}\ \{\,\textit{``Match''} \Rightarrow M_{n_i}(x_i, t_i)\} \\ & \mathsf{where}\ x_i \Rightarrow t_i\ \mathsf{in}\ t \end{aligned}$$

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 an extended regular query.

Example 6.3 Consider the query Q:

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

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:

$$M_{DB}(DB) = \{``1'' \Rightarrow S_t(DB)\} \qquad // \ ``DB'' \Rightarrow DB \text{ not necessary}$$
$$S_t(DB) = \quad \mathsf{select}\ \{\,\textit{``Match''} \Rightarrow M_t(t)\}$$

```
                where $R \Rightarrow t$ in $DB$
$M_t(t) =$      $\{``1'' \Rightarrow S_{t_1}(t), ``2'' \Rightarrow S_{t_2}(t)\}$ // $``t'' \Rightarrow t$ not necessary
$S_{t_1}(t) =$  select { "Match" $\Rightarrow M_{t_1}(t_1)\}$
                where $R_1 \Rightarrow t_1$ in $t$
$S_{t_2}(t) =$  select { "Match" $\Rightarrow M_{t_2}(t_2)\}$
                where $R_2 \Rightarrow t_2$ in $t$
$M_{t_1}(t_1) = \{``t_1'' \Rightarrow t_1\}$
$M_{t_2}(t_2) = \{``t_2'' \Rightarrow t_2\}$
```

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:

```
$Q_r(DB) = M_{DB}(DB) =$
  {"1"$\Rightarrow$ select { "Match" $\Rightarrow$ { "1"$\Rightarrow$ select { "Match" $\Rightarrow \{``t_1'' \Rightarrow t_1\}\}$
                                          where $R_1 \Rightarrow t_1$ in $t$,
                                   "2"$\Rightarrow$ select { "Match" $\Rightarrow \{``t_2'' \Rightarrow t_2\}\}$
                                          where $R_2 \Rightarrow t_2$ in $t\}\}$
          where $R \Rightarrow t$ in $DB\}$
```

□

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 subquery $Q'(t_1, \ldots, t_k)$ 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:

```
select $E$
where $P_1$ in $s_1, \ldots, P_n$ in $s_n$
```

By abuse of notation we will 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:

$$\begin{array}{ll} \text{"}i\text{"} \Rightarrow s_{n_t} & \text{in } m_s, \\ \text{"}\mathit{Match}\text{"} \Rightarrow m_t & \text{in } s_t \\ \text{"}t\text{"} \Rightarrow t & \text{in } m_t \end{array}$$

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 $\{x \Rightarrow DB\}$ in $x \Rightarrow \_$ 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 $Q_s(DB, P)$ is:

```
select {"A" ⇒ {"B" ⇒ t_1, "C" ⇒ t_2}}
where "1" ⇒ s_t in P                  // translation of R ⇒ t in DB
      "Match" ⇒ m_t in s_t
      "1" ⇒ s_t1 in m_t               // translation of R_1 ⇒ t_1 ∈ t
      "Match" ⇒ m_t1 in s_t1
      "t_1" ⇒ t_1 in m_t1
      "2" ⇒ s_t2 in m_t               // translation of R_2 ⇒ t_2 ∈ t
      "Match" ⇒ m_t2 in s_t2
      "t_2" ⇒ t_1 in m_t2
```

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

Summarizing, we have:

Theorem 6.5 *Given a selection 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 an extended regular query.*

Proof: 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 $Q_r(DB) = M_{DB}(DB)$. Then:

$$
\begin{aligned}
Q_s(Q_r(DB)) = \mathsf{select}\ & E(t_1, t_2) \\
\mathsf{where}\ & ``1'' \Rightarrow s_t\ \mathsf{in}\ M_{DB}(DB) \\
& \quad \text{“}\mathit{Match}\text{”} \Rightarrow m_t\ \mathsf{in}\ s_t \\
& \quad \ldots \text{ the other generators}
\end{aligned}
$$

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

$$
\begin{aligned}
Q_s(Q_r(DB)) = \mathsf{select}\ & E(t_1, t_2) \\
\mathsf{where}\ & \text{“}\mathit{Match}\text{”} \Rightarrow m_t\ \mathsf{in}\ S_t(DB) \\
& \quad \ldots \text{ the other generators}
\end{aligned}
$$

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

$$
\begin{aligned}
Q_s(Q_r(DB)) = \mathsf{select}\ & E(t_1, t_2) \\
\mathsf{where}\ & R \Rightarrow t\ \mathsf{in}\ DB \\
& \quad ``1'' \Rightarrow s_{t_1}\ \mathsf{in}\ M_t(t) \\
& \quad \text{“}\mathit{Match}\text{”} \Rightarrow m_{t_1}\ \mathsf{in}\ s_{t_1} \\
& \quad ``t_1'' \Rightarrow t_1\ \mathsf{in}\ m_{t_1} \\
& \quad ``2'' \Rightarrow s_{t_2}\ \mathsf{in}\ M_t(t) \\
& \quad \text{“}\mathit{Match}\text{”} \Rightarrow m_{t_2}\ \mathsf{in}\ s_{t_2} \\
& \quad ``t_2'' \Rightarrow t_1\ \mathsf{in}\ m_{t_2}
\end{aligned}
$$

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}:

$$
\begin{aligned}
Q_s(Q_r(DB)) = \mathsf{select}\ & E(t_1, t_2) \\
\mathsf{where}\ & R \Rightarrow t\ \mathsf{in}\ DB \\
& \quad \text{“}\mathit{Match}\text{”} \Rightarrow m_{t_1}\ \mathsf{in}\ S_{t_1}(t) \\
& \quad ``t_1'' \Rightarrow t_1\ \mathsf{in}\ m_{t_1} \\
& \quad \text{“}\mathit{Match}\text{”} \Rightarrow m_{t_2}\ \mathsf{in}\ S_{t_2}(t) \\
& \quad ``t_2'' \Rightarrow t_1\ \mathsf{in}\ m_{t_2}
\end{aligned}
$$

Next, $S_{t_1}(t) = \mathsf{select}\ \{\text{“}\mathit{Match}\text{”} \Rightarrow M_{t_1}(t_1)\}\ \mathsf{where}\ R_1 \Rightarrow t_1\ \mathsf{in}\ t$, 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}:

$$
\begin{aligned}
Q_s(Q_r(DB)) = \mathsf{select}\ & E(t_1, t_2) \\
\mathsf{where}\ & R \Rightarrow t\ \mathsf{in}\ DB
\end{aligned}
$$

$R_1 \Rightarrow t_1$ in t
"$t_1'' \Rightarrow t_1$ in $M_{t_1}(t_1)$
$R_2 \Rightarrow t_2$ in t
"$t_2'' \Rightarrow t_1$ in $M_{t_2}(t_2)$

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 subblock of a select − where block becomes unnecessary if some other subblock of the same select − where block is empty (i.e. has no matching).
2. A variable x or t bound in some select − 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 selection 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 selection 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 subblock 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 subblocks 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 subblocks 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': intuitively, the subblock of B dominated by n' exists (i.e. is nonempty) iff there exists at least one matching under n' for which the same subblock 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 subblock 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 is n is a leave in one of B's subblocks. 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: the subblock 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 subblock 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 $n'_1 \overset{“x”}{\to} n'_2$ or $n'_1 \overset{“t”}{\to} n'_2$ 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:

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

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 subblocks in the pattern tree shown in Figure 18. We show in Figure 20 (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 20). Both m_t nodes have two children[7], 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 subblock of B exists, and similarly for t_2. On the next level (with the math-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 subblocks of B it sees exists iff both the subblocks for t_1 and for t_2 exists. Since t's subblocks 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 20 (a)), the other is empty (the right half). The emptiness information for each block instantiation is then distributed to all subblocks. 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

[7]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.

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 20 (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.

$\Box$

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.

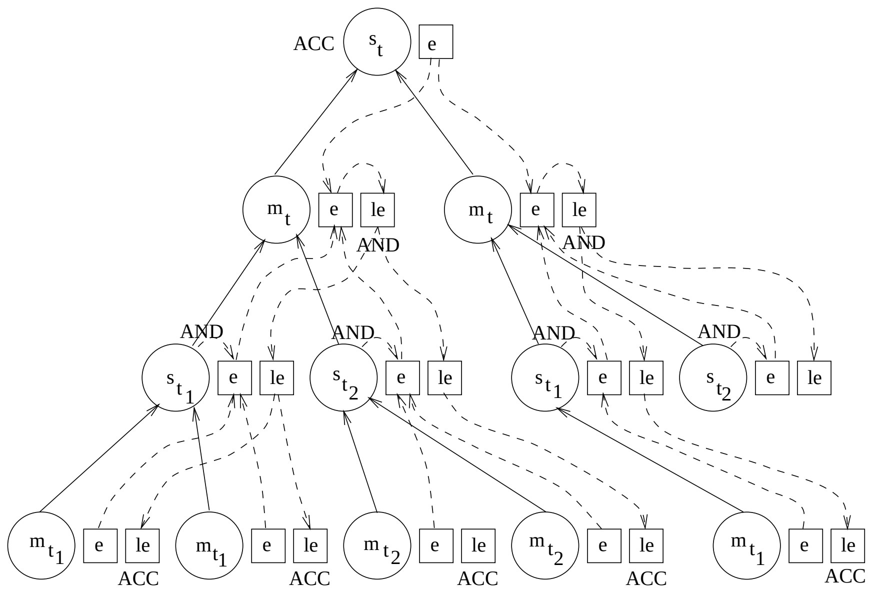
6.3 Solving the AGAP for a Distributed Database

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.

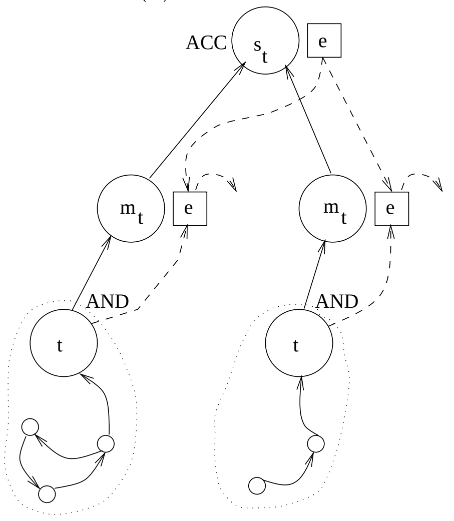
However in our case we have to solve AGAP's of a particular form: in which the AND nodes do not belong to cycles, and in which their outdegree is "small". Specifically, we call an alternating graph G *AND-acyclic* iff none of its AND nodes belong to a cycle. Next, for a AND-acyclic graph G we define 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.

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:



(a)



(b)

Figure 20: 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'.

```
Algorithm : Distributed-Evaluation-AGAP
Input :     A alternating, AND-acyclic graph $G$
                with AND-outdegree $\delta$
                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 $\alpha$ compute the input-output graph $IO_\alpha$ (see text)
            Step 2  Send all $IO_\alpha$, $\alpha = 1, m$ 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 21: Distributed AGAP.

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 $s_1 \to s_2$ 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. $\Box$

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. $\Box$

In fact all the information about connectivity can be found in those edges $s \to s'$ of G' in which s is a singleton set. Indeed there is an edge $s \to s'$ in G' iff $\forall n \in s$ there exists an edge $\{n\} \to s''$ in G' with $s'' \subseteq s'$. This observation enables us to derive an efficient distributive algorithm for the AGAP of a AND-acyclic graph: this is shown in Figure 21. 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 selection queries on distributed databases. For a selection 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 extended regular queries have $\delta = 1$.

Theorem 6.11 *Let Q be a selection query with b blocks and with block fragmentation δ, and DB be a distributed database. Let n be the number of crosslinks, 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 (same as Algorithm *Distributed-Evaluation-$\mathcal{C}$* , the only difference is 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 subblocks, including the fragment dominated by the current match-set node. For example in Figure 20 (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 subblock 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 subblock 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 subblock (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 Regular 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 Regular 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 Theorem 5.10 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 Theorem 5.10 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 $DB_1' := \mathsf{rec}\ (DB_1; \Delta)$. 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 rec rather than[8] $+\!\!+$. Finally we can maintain V_1 as $V_1' = Q^{dec}(\mathsf{rec}\ (DB_1; \Delta)) = \mathsf{rec}\ (V_1; Q^{dec}(\Delta))$.

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 Selection Queries Finally we consider selection 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 regular 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 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 $V' = Q_s((P_1')^{acc})$.

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

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 regular path expressions, graph constructors, and nested queries. In their most general forms, the algorithms cover two incomparable classes of queries: the class $\mathcal{C}$, and the selection queries. 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, selection 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 *ICDT*, 1997.

[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 *ICDT*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.

[BDHS96a] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, 1996.

[BDHS96b] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. Technical Report 96-09, University of Pennsylvania, Computer and Information Science Department, February 1996.

[BDS95] Peter Buneman, Susan Davidson, and Dan Suciu. Programming constructs for unstructured data. In *Proceedings of DBPL'95*, Gubbio, Italy, September 1995.

[Bun97] Peter Buneman. Tutorial: Semistructured data. In *PODS*, 1997.

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

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

[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 *VLDB*, 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*, March 1995.

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

[Suc96] Dan Suciu. Query decomposition for unstructured query languages. In *VLDB*, September 1996.

[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.

A Appendix

For completeness, we revise here the definition of bisimulation, which we take as semistructured database equality [BDS95].

Given two nodes u, v in some graph G, we denote with $u \overset{\varepsilon^*}{\to}\overset{a}{\to} v$ 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 A.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 $u \overset{\varepsilon^*}{\to}\overset{a}{\to} v$ in G, then there exists some path $u' \overset{\varepsilon^*}{\to}\overset{a}{\to} v'$ (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 $u' \overset{\varepsilon^*}{\to}\overset{a}{\to} v'$ in G', then there exists some path $u \overset{\varepsilon^*}{\to}\overset{a}{\to} v$, and $(v, v') \in R$.*

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

This is *not* is not the same as weak bisimulation in process algebra (see for example [Mil89]): see [BDHS96b] for a discussion.

The definition applies mutatis mutandis to graphs with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$, Definition 5.1:

Definition A.2 *Given two graphs $G, G' \in \mathit{Tree}^{\mathcal{X}}_{\mathcal{Y}}$, a* **bisimulation** *from G to G' is a binary relation $R \subseteq \mathit{Nodes}(G) \times \mathit{Nodes}(G')$ such that:*

1. *For every input marker $X \in \mathcal{X}$, $(\mathit{Input}_X(G), \mathit{Input}_X(G')) \in R$.*
2. *Condition 2 in Definition A.1 is satisfied.*
3. *Condition 3 in Definition A.1 is satisfied.*
4. *Whenever $(u, u') \in R$ and there exists a path $u \overset{\varepsilon^*}{\to} v$ in G with v a output node labeled with marker Y, then there exists a path $u' \overset{\varepsilon^*}{\to} v'$ in G' such that v' is a output labeled Y and $(v, v') \in R$.*
5. *Similarly, with the roles of G and G' reversed.*