Abstract

Recently, several query languages have been proposed for querying information sources whose data is not constrained by a schema, or whose schema is unknown. Examples include: LOREL (for querying data combined from several heterogeneous sources), W3QS (for querying the World Wide Web); and UnQL (for querying unstructured data).

The natural data model for such languages is that of a rooted, labeled graph. Their main novelty is the ability to express queries which traverse arbitrarily long paths in the graph, typically described by a regular expression. Such queries however may prove difficult to evaluate in the case when the data is distributed on several sites, with many edges going between sites. A typical case is that of a collection of WWW sites, with links pointing freely from one site to another (even forming cycles). A naive query shipping strategy may force the query to migrate back and forth between the various sites, leading to poor performance (or even non-termination). We present a technique for query decomposition, under which the query is shipped exactly once to every site, computed locally, then the local results are shipped to the client, and assembled here into the final result. This technique is efficient, in that (a) only data which is part of the final result is shipped from the data sites to the client site, and (b) the total work done locally at all sites does not exceed that needed for computing the (unoptimized) query on a centralized version of the database.

We also show that the query decomposition technique can be adapted to derive a simple view maintenance method, for two forms of updates which we introduce for the graph data model.

1 Introduction

Several database query languages have been proposed recently, in which the data model is that of a labeled tree, or, more generally, a labeled, rooted graph. Loretl [QRS+95] was designed in conjunction with the Tsimmis project [PGMW95], to query data from information sources which do not impose a rigid structure, or whose structure is not completely known. UnQL [BDS95, BDHS96a], was motivated by the need to query self-describing databases, like the ACeDB databases popular among biologists, and also for data sources with unknown structure. Authors of both languages suggest their suitability to query information sources on the World-Wide Web. W3QS [KS951 is a query language specifically designed for querying the World-Wide Web. It was motivated by the fact that today's web search engines are restricted to content based queries, which select a single page based on its content: W3QS was designed to ask structure based queries, addressing the hypertext organization itself. Another system developed for querying hypertext structures is presented in [BK90].

The main common feature of these languages is their ability to follow arbitrarily long sequences of links. We illustrate this feature by giving a typical query, which we will use throughout this paper.

Example 1.1 Suppose that we want to fetch all publications from the Computer Science Department at the University of California at San Diego, whose home page is http://www.ucsd.edu. The html structure we need to search is illustrated in Figure 1. It is convenient to view html data as a rooted, directed graph: nodes correspond to pages, edges to links, and each page is a set of all its outgoing links. The prob-

1In Figure 1, and throughout the paper, we will ignore the node content: thus our trees will have labeled edges, not nodes. In the case of html data we justify our choice by the fact that we focus on structural, rather than content based queries. See also [BDHS96a] for more justification of this model.
lem in expressing this query in a relational or object-oriented query language is that we don’t know in advance how many links to follow from the root to the CS-Department link, and from here to the Papers link. In a slightly modified UnQL syntax this query is expressed as:

```
select "Papers".t
where *."CS-Department"... *."Papers".t in DB
```

Here DB stands for the “database”, in this example the http://www.ucsd.edu site. The symbol - denotes any label, while * placed after some label means that it can be repeated 0 or more times. Thus *.. means any sequence of labels. In short, the query starts from the root, follows all paths to an edge labeled “CS Department”, from there follows all paths to an edge labeled “Papers”, and returns the set of all subtrees thus found. Of course, the same query can be applied to other university sites, to retrieve papers published in their Computer Science Departments.

This example illustrates the main feature separating query languages like UnQL, LOREL, W3QS from relational or object-oriented query languages: their ability to follow arbitrarily long sequences of links. Typically such sequences are described by regular expressions. In particular we can compute the transitive closure of the link relation. Following [BDHS96a] we will call such languages unstructured query languages. Their natural data model is that of a rooted, directed graph, with labels attached to vertices, or edges, or both. Traditional relational databases are captured by this model as the special case of trees of a fixed depth ([BDHS96a], see also Section 3 in this paper). Over relational databases however, the expressive power of such languages is no more powerful than that of the relational algebra [BDHS96a].

In this paper we discuss two problems in connection with such languages: query decomposition and view maintenance. We describe these two problems next.

**Query decomposition** Information sources sometimes reside on a number of different sites. The database fragment in any one site may have a large number of links leading to other sites. To illustrate, suppose we want to consider UCSD’s web site in Example 1.1, in conjunction with the web site of the San Diego Supercomputer Center, http://www.sdsc.edu (Figure 2). A typical query Q may start at the UCSD web site, but then will traverse back and forth between the two sites, as it traverses sequences of links. E.g. the “CS-Department” may have links to projects at the SDSC site, while from there we may follow links back to the UCSD site, etc: at any point, we may encounter a “Papers” link. If query shipping is used for query evaluation, this implies that the query has to migrate between the two sites an arbitrary large number of times. The query decomposition problem requires the query Q to be decomposed into two queries which can be computed independently on the fragments DB1 and DB2 of the database residing at the two sites, and their results combined at the client’s site to yield Q(DB). Of course, this problem can be solved best when additional knowledge about the data distribution is available, e.g. that the “CS-Department” link is unique and that it resides at the UCSD site. But our purpose here is to see how much we can accomplish without any knowledge about the data distribution: this is useful, e.g. when browsing an unfamiliar information source. We will describe a technique which translates any query Q in a certain fragment of an unstructured query language, into a query Q’ which is decomposable, meaning that Q’(DB) can be computed by computing Q’(DB1) and Q’(DB2) independently, then gluing the two results at the client’s site. Moreover, this decomposition is efficient: computing Q’(DB1) and Q’(DB2) is no more expensive than computing Q(DB), using some naive evaluation strategy, when DB is centralized. However an optimized evaluation of Q(DB) could be more efficient on a centralized version of DD than that of computing Q’(DB1) and Q’(DB2) separately.

**View maintenance** A view V is just the result of some query Q applied to the database: V = Q(DB). The view is materialized when the result V is stored at the client site for future use. The view maintenance problem consists in finding efficient techniques for updating the view V at the client site when the database is incrementally updated, say DB’ := DB ++ Δ, where Δ is much “smaller” than DB (++ defined in Section 3: for the case of relational databases represented as trees, insertion of an element into a set can be modeled as a special case of ++). Ideally one would like the view update to depend only on the old view and
the increment, $V' := f(V, \Delta)$, not on $DB$. View main-
tenance techniques for relational databases have been
extensively discussed, see e.g. [GL95] for a list of ref-
ences. In the case of unstructured data however, we
face an additional problem: namely that of informing
the client site where the update has taken place. The
"where" information is taken for granted in the case
of a relational query language. E.g. consider a rela-
tional database with three relations $R_1, R_2, R_3$, and
and the view $V = R_1 \cup \sigma_{\text{where}=3}(R_2)$. In classic view main-
tenance algorithms the view will be updated differently,
depending on whether $R_1$, $R_2$ or $R_3$ has been updated.
E.g. when $R_2 := R_2 \cup \Delta$, then the view becomes
$V' := V \cup \sigma_{\text{where}=3}(\Delta)$, but when $R_3$ gets updated the view
does not need to be changed. In an "unstructured"
database there is no static partition of the data into
distinct relations, but the partition is done dynami-
cally, when the view is computed. The query in Ex-
ample 1.1 defines a view which logically partitions the
database into three regions, as in Figure 3. The regu-
lar expression occurring in this query has an equivalent
automaton with three states: the three regions corre-
spond precisely to these states. View maintenance
will be done differently after an update in region $R_1$ than
after updates in regions $R_2$ or $R_3$ respectively. Of
course a view defined by another query may partition
$DB$ in a different way. In this paper we develop alge-
bric techniques which transform any query $Q$ in a cer-
tain fragment of an unstructured query language into a
query $Q'$ (the same as in the query decompo-
sition problem), such that (1) $Q(DB)$ can be computed eas-
ily from $Q'(DB)$, (2) the computation of $Q'(DB)$ is no
more expensive than a naive computation of $Q(DB)$,
and (3) $Q'(\Delta)$ "encapsulates" the information about
which region has been affected by the update, and,
therefore: $Q'(DB + + \Delta) = Q'(DB) + + Q'(\Delta)$.

1.1 Relation to previous work

We choose to describe our techniques in the context
of the query language UnQL [BDHS96a, BDHS96b].
The language relies on a data model of rooted, labeled
graphs, which was first described in [BDS95]. The
same paper describes bisimulation between rooted, la-
beled graphs, showing how this data model subsumes
both the relational and the nested relational one. Fi-
ally it introduces the basic algebraic operations asso-
ciated to this data model, including tree concatenation
(denoted ++ in this paper), and $\text{vext}$ ("vertical" $\text{vext}$),
observing that $\text{vext}$ works nicely in conjunction with
cyclic structures. The language UnQL is introduced
in [BDHS96a, BDHS96b], as a declarative language
with pattern matching, and is shown to be equivalent
to an algebraic language UnCAL, centered around ++
and a generalized version of $\text{vext}$ called $\text{gext}$ ("gen-
eralized" $\text{vext}$). [BDHS96a, BDHS96b] prove a num-erv of algebraic laws for $\text{gext}$ which are intended to
be used for query optimizations. In this paper we
use these laws and additional ones, dealing with cyclic
data, in order to derive our query decomposition tech-
niques. We also need to add some additional opera-
tions to UnCAL: namely we consider labeled graphs
with several roots as first class objects in the language
(much like the n-trees in [Cou83, pp 134]), and extend
UnCAL with $\text{graph juxtaposition}$, see Section 4. The
equations associated to these new operations, much in
the spirit of [Cou83, pp 135], are simpler however than
those for ++ and $\text{vext}$.

1.2 Applicability, assumptions and limitations

Query language Although we describe our tech-
niques in the context of the query language
UnQL [BDS95, BDHS96a], they apply equally well for
decomposing queries in (certain fragments of) LOREL
or W3QS. It is important to notice however that dur-
ing both query decomposition and view main-
tenance, we use in a critical way some of the particular language
constructs in UnQL, such as $\text{tree concatenation}$, $\text{vext}$ $\text{vext}$
(denoted $@$ in [BDHS96a]). Moreover we note that our
techniques apply to some UnQL queries which are not
expressible in LOREL or W3QS.

Restrictions on the query language Our query
decomposition technique works only for UnQL queries
satisfying two restrictions. The first requires the query
to be "monotone". In particular we cannot handle
set difference, or incremental deletions. This is an
expected limitation, because our data model has a set
semantics: in particular flat trees are just sets. For the
case of relational databases, it is known [GL95] that
bag semantics rather than set semantics is needed in
order to do algebraic view maintenance for deletions.
The second restriction requires the query to be join-
free. This limitation is far less severe than it sounds,
because in unstructured query languages the focus is
on queries traversing long link sequences, which are
join free: as in object oriented databases, most joins
are replaced by link traversals. In fact all queries
in W3QS [K95] are join-free, while most interesting
LOREL and UnQL examples [QRS+95, BDHS96a] are
join-free too.

No knowledge about data distribution Our query
decomposition method works without assuming
any knowledge about how the data is distributed on
the sites. While sometimes this may be useful, we
consider it to be the most serious limitation of our approach. In most cases minor knowledge about the data sources could help one decompose a query much better than we currently do. As part of future work, we plan to investigate how knowledge about data sites can be incorporated into our query decomposition method.

Updates Our view maintenance techniques work for two kinds of updates: insertions and replacements. An insertion, in notation $DB + \Delta$, means that some new subtree $\Delta$ is inserted at some particular node $v$ in $DB$ (Section 3); the root of $\Delta$ will be “merged” with $v$. This is a monotone operation, in that the resulting database has at least as many edges as the original one. The second kind of update, replace, allows us to replace a subtree rooted at a given node with another tree. This is not a monotone operation and can, to some extent, model deletions, like those in a relational databases.

Update notification Our view maintenance technique requires that the server informs the client whenever a page (or, in general, a node in the graph) is updated. We refer the reader to [BD96, DB96] for update notification techniques for the WWW.

Site selection and query capabilities We do not address the problem of site selection. Instead, we always assume that the database is stored at a fixed, known, relatively small number of sites, say $s_1, \ldots, s_k$, which may have links between them. This assumption could be easily enforced, in the case of web sites, by simply ignoring all links pointing outside the set $s_1, \ldots, s_k$. Also, we assume no knowledge about the semantics of data stored at these sites, or about their query capabilities: in reality some sites may offer only restricted access, e.g. using a keyword search. For the case of conjunctive queries over relational databases, [LMSS96, LRU96] discuss techniques which can efficiently select the relevant sites and also use the limited query capabilities of such sites.

2 An Example

We illustrate next query decomposition. Assume that the database $DB$ is distributed on two different sites $s$ and $s'$, with arbitrary many links between them: then a path of the form

$$... \bullet \text{"CS-Department"...} \bullet \text{"Papers"...}$$

may travel back and forth between $s$ and $s'$ an arbitrary number of times. Since we assume no a priori knowledge about how the data is structured and/or partitioned on the two sites, we should be prepared to deal with more than one “CS-Department” links, with any number of “Papers” links, and with any possible distribution of these links on the two sites. To decompose the query we need to have a list of all input and output links at each site (Figure 4 (a)). Intuitively, our method starts by decomposing the query $Q$ into the three queries $Q_1, Q_2, Q_3$ described above, and then applies each of these three queries to every entry point at each site; this is illustrated by the six inputs $Q_1(X), \ldots, Q_3(Y)$ at the site $s$ and the six inputs $Q_1(U), \ldots, Q_3(V)$ at the site $s'$ in Figure 4 (b). Now we observe that each of these queries logically partitions the database into up to three regions ($Q_2$ defines only two, while $Q_3$ only one). We want to tag each output link with the corresponding region name where it was found. But note that the three partitions are independent, and each output link may be “found” in more than one region, say in $R_3$ by $Q_1$, and in $R_2$ by $Q_2$. Hence we replicate each output link 3 times, once for each region where it may be found. This is illustrated by the six outputs at sites $s$ and $s'$

**1.** If $\Delta$ is inserted before any “CS-Department” edge (in region $R_1$ of Figure 3), then $\Delta' = Q_1(\Delta)$, where $Q_1(\Delta)$ is:

$$\text{select } \text{"Papers".}t$$

$$\text{where...} \bullet \text{"CS-Department"...} \bullet \text{"Papers".}t$$

$$\text{in } \Delta$$

**2.** If $\Delta$ is inserted between a “CS-Department” edge and a “Papers” edge (region $R_2$ of Figure 3), then $\Delta' = Q_2(\Delta)$, where $Q_2(\Delta)$ is:

$$\text{select } t$$

$$\text{where...} \bullet \text{"Papers".}t$$

$$\text{in } \Delta$$

**3.** Finally, if $\Delta$ is inserted after a “CS-Department” and after a “Papers” edge, then $\Delta' = Q_3(\Delta) = \Delta$.

While our informal description here sounds rather procedural, the technique, as described further in this paper, is fully algebraic.

We illustrate next query decomposition. Assume that the database $DB$ is distributed on two different sites $s$ and $s'$, with arbitrary many links between them: then a path of the form

$$... \bullet \text{"CS-Department"...} \bullet \text{"Papers"...}$$

may travel back and forth between $s$ and $s'$ an arbitrary number of times. Since we assume no a priori knowledge about how the data is structured and/or partitioned on the two sites, we should be prepared to deal with more than one “CS-Department” links, with any number of “Papers” links, and with any possible distribution of these links on the two sites. To decompose the query we need to have a list of all input and output links at each site (Figure 4 (a)). Intuitively, our method starts by decomposing the query $Q$ into the three queries $Q_1, Q_2, Q_3$ described above, and then applies each of these three queries to every entry point at each site; this is illustrated by the six inputs $Q_1(X), \ldots, Q_3(Y)$ at the site $s$ and the six inputs $Q_1(U), \ldots, Q_3(V)$ at the site $s'$ in Figure 4 (b). Now we observe that each of these queries logically partitions the database into up to three regions ($Q_2$ defines only two, while $Q_3$ only one). We want to tag each output link with the corresponding region name where it was found. But note that the three partitions are independent, and each output link may be “found” in more than one region, say in $R_3$ by $Q_1$, and in $R_2$ by $Q_2$. Hence we replicate each output link 3 times, once for each region where it may be found. This is illustrated by the six outputs at sites $s$ and $s'$

$\text{In this case the correct view update is of the form } V' := V \cup \Delta'; \text{ see Section 8.}$
in Figure 4 (b). In the next step the results of the two independent computations are shipped to the client, where they are combined, by gluing the outputs with their “corresponding” inputs.

3 Data Model

As mentioned earlier, we adopt the data model in [BDS95, BDHS96a]. Unlike the Tsimmis data model [PGMW95, QRS+95] it does not have object id’s, and has two additional features, markers and e-edges, which we use in a critical way for both query decomposition and view maintenance. We briefly review here the main ideas and refer the reader to [BDHSSb] for a detailed description of the data model.

Rooted graphs Let Label be the universe of all labels: it includes all strings, numbers, booleans, etc. A database is modeled as a rooted graph (i.e. a graph with a distinguished node called the root), whose edges are labeled with elements from Label ∪ {e}. Here e is a special label, denoting an “empty” symbol: whenever two vertices v, v’ are connected by an e edge, the intended meaning is that all edges emerging from v’ should also emerge from v, see Figure 5.

Trees Trees form a particularly interesting subset of the rooted graphs, and they suffice to represent sets and records. Figure 6 contains an example of a relational database and its representation as a tree. The following is a syntax for trees:

\[
\text{Tree ::= \{} \mid \{\text{Label} \Rightarrow \text{Tree}\} \mid \text{Tree} \cup \text{Tree}
\]

Abbreviating \(\{a_1 \Rightarrow t_1, \ldots, a_n \Rightarrow t_n\}\) for \(\{a_1 \Rightarrow t_1\} \cup \ldots \cup \{a_n \Rightarrow t_n\}\), and \(\{a\}\) for \(\{a \Rightarrow \{\}\}\), the example in Figure 6 is written as:

\[
\text{t1} = \{a \Rightarrow \{e\}, b \Rightarrow \{d\}, X \Rightarrow \{\}\, Y \Rightarrow \{\}\}
\]

\[
\text{t2} = \{a \Rightarrow \{c\}, b \Rightarrow \{\}, Z \Rightarrow \{\}\}
\]

We emphasize that this data model has set semantics. E.g. the trees \(\{a, b \Rightarrow \{c\}\, b \Rightarrow \{\}\}\) and \(\{a, b \Rightarrow \{\}\}\) are considered equal.

Markers In addition to the edge labels, some of the leaves of a graph are allowed to be labeled with special symbols, denoted \(X, Y, \ldots\), called markers. Unlike labels, markers are not part of the information content of the database, but are used to control (1) where updates take place, and (2) how to connect fragments of a distributed database. They share some similarities with the object id’s used in the Tsimmis data model [PGMW95, QRS+95]. Markers allow us to define the concatenation operation \(++x\): given two graphs \(t_1, t_2\) and a marker \(X\), \(t_1 \, ++x \, t_2\) denotes the database obtained by drawing e edges from all leaves labeled X in \(t_1\) to the root of \(t_2\). All occurrences of the old marker \(X\) in \(t_1\) disappear in \(t_1 \, ++x \, t_2\). But all other markers in \(t_1\) remain in \(t_1 \, ++x \, t_2\), as well as all markers from \(t_2\), see Figure 7.

Graphs with \(m\) inputs, \(n\) outputs To capture the connection between fragments of a distributed database, we generalize from single input (the root of the tree) to \(m\) inputs. Also we call the markers on the leaves the outputs of the tree, thus reaching the notion of a tree (graph) with \(m\) inputs and \(n\) outputs. Formally, let \(X = \{X_1, \ldots, X_m\}\) and \(Y = \{Y_1, \ldots, Y_n\}\) be two finite sets of markers, \(m \geq 0, n \geq 0\). A database
with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$ is a graph with edges labeled with elements from $\text{Label} \cup \{\varepsilon\}$ and in which some leaves may be labeled with markers in $\mathcal{Y}$ (as before), and with $m$ distinguished roots associated to $X_1, \ldots, X_m$. We extend the syntax for trees to that of trees with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$, $\text{Tree}_{\mathcal{Y}}$, and to trees with a single, anonymous input and with outputs $\mathcal{Y}$, $\text{Tree}_{\varepsilon}$:

\[
\text{Tree}_{\mathcal{Y}} := (X_1 := \text{Tree}_{\mathcal{Y}_1} ; \ldots ; X_m := \text{Tree}_{\mathcal{Y}_m}) \\
\text{Tree}_{\varepsilon} := \{} | (\text{Label} \Rightarrow \text{Tree}_{\mathcal{Y}}) | \text{Tree}_{\mathcal{Y}} \cup \text{Tree}_{\varepsilon} | Y_i (j = 1, n)
\]

Figure 8 contains an example of a graph with inputs $X_1, X_2, X_3$ and outputs $Y_1, Y_2$, which is written as $(X_1 := \{a \Rightarrow Y_1 \cup \{b \Rightarrow Y_2\}; X_2 := \{b \Rightarrow \{Y_1 \cup \{b \Rightarrow \{Y_2\}\}\}, a\}; X_3 := \{c \Rightarrow \{a, b \Rightarrow Y_1\}\})$. By convention, there exists a single graph with inputs $\mathcal{X} = \emptyset$, namely the null graph, denoted $\{}$ (no nodes, no edges; this is different from the empty set, $\{\}$, which has one node, no edges). We extend the $+$ operation as follows. For $t_1 \in \text{Tree}_{\mathcal{X}}$ and $t_2 \in \text{Tree}_{\mathcal{Y}}$, $t_1 +_{\varepsilon} t_2$ is the tree in $\text{Tree}_{\mathcal{Y}}$ obtained by drawing an $\varepsilon$ edge from every output $Y_i$ in $t_1$ to the input $Y_i$ in $t_2$, $i = 1, n$.

Recursive definitions For $t \in \text{Tree}_{\mathcal{X}}$, $\text{rec}_{\mathcal{X}} t$ denotes a graph obtained by adding an $\varepsilon$-edge from every output $X_i$ to the input $X_i$, $i = 1, m$ (Figure 9). Intuitively $\text{rec}_{\mathcal{X}} t$ is $t +_{\varepsilon} t +_{\varepsilon} t +_{\varepsilon} \ldots$ E.g. $\text{rec}_{\mathcal{X}} (X := \{a \Rightarrow X\})$ defines a loop labeled $a$.

Equality The notion of equality on rooted graphs is that of bisimilarity [BDHS96a, BDS95]. In a nutshell, two rooted graphs are bisimilar if, after (possible infinite) unfolding, $\varepsilon$-edge removal, and duplicate subtree elimination at each node, the two resulting trees are equal. When restricted to tree representations of sets and records, as in Figure 6, the bisimilarity relation is exactly the set and/or record equality.

$\varepsilon$-Edges $\varepsilon$-Edges are introduced for convenience of notation: most operations, like $+_{\varepsilon}$ or $\text{rec}_{\varepsilon}$, are easier defined in terms of $\varepsilon$ edges, than without them. On the other hand the reader may have noticed that, in some sense, they are redundant: every rooted graph with $\varepsilon$ edges can be shown to be bisimilar to a rooted graph without $\varepsilon$ edges [BDHS95, BDHS96a]. But this only works when the graphs have no markers. E.g. the tree $(\{a\} \cup X$ represented with an $\varepsilon$ edge as $(\{a\} \cup \{b \Rightarrow X\})$, is not equivalent to any tree without $\varepsilon$ edges.

Notation As in [BDHS96a], by abuse of notation we call "trees" all graphs, even when they have cycles. Thus $\text{Tree}_{\varepsilon}$ denotes the set of all databases with a single anonymous input and outputs $\mathcal{Y}$, and $\text{Tree}_{\mathcal{Y}}$ that of all databases with inputs $\mathcal{X}$, outputs $\mathcal{Y}$. When $\mathcal{Y} \subseteq \mathcal{Y}'$, then $\text{Tree}_{\mathcal{Y}} \subseteq \text{Tree}_{\mathcal{Y}'}$ and $\text{Tree}_{\mathcal{X}} \subseteq \text{Tree}_{\varepsilon}$.

Canonical form Every database $DB$ in $\text{Tree}_{\varepsilon}$ can be expressed (not necessarily uniquely) as:

\[
X_1 +_{\varepsilon} \mathcal{X} \text{rec}_{\mathcal{X}} ((X_1 := t_1; \ldots ; X_m := t_m))
\]

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

4 Representing Distributed Databases

We will illustrate how a distributed database can be represented in our notation, using the example in Figure 10, where $DB$ is stored on two sites $s, s'$. We start by cutting the cross links, and inserting marker sites $s_5, s'_5$. The notion of equality on rooted graphs is that of bisimilarity [BDHS96a, BDS95]. In a nutshell, two rooted graphs are bisimilar if, after (possible infinite) unfolding, $\varepsilon$-edge removal, and duplicate subtree elimination at each node, the two resulting trees are equal. When restricted to tree representations of sets and records, as in Figure 6, the bisimilarity relation is exactly the set and/or record equality.

$\varepsilon$-Edges $\varepsilon$-Edges are introduced for convenience of notation: most operations, like $+_{\varepsilon}$ or $\text{rec}_{\varepsilon}$, are easier defined in terms of $\varepsilon$ edges, than without them. On the other hand the reader may have noticed that, in some sense, they are redundant: every rooted graph with $\varepsilon$ edges can be shown to be bisimilar to a rooted graph without $\varepsilon$ edges [BDHS95, BDHS96a]. But this only works when the graphs have no markers. E.g. the tree $(\{a\} \cup X$ represented with an $\varepsilon$ edge as $(\{a\} \cup \{b \Rightarrow X\})$, is not equivalent to any tree without $\varepsilon$ edges.

Notation As in [BDHS96a], by abuse of notation we call "trees" all graphs, even when they have cycles. Thus $\text{Tree}_{\varepsilon}$ denotes the set of all databases with a single anonymous input and outputs $\mathcal{Y}$, and $\text{Tree}_{\mathcal{Y}}$ that of all databases with inputs $\mathcal{X}$, outputs $\mathcal{Y}$. When $\mathcal{Y} \subseteq \mathcal{Y}'$, then $\text{Tree}_{\mathcal{Y}} \subseteq \text{Tree}_{\mathcal{Y}'}$ and $\text{Tree}_{\mathcal{X}} \subseteq \text{Tree}_{\varepsilon}$.

Canonical form Every database $DB$ in $\text{Tree}_{\varepsilon}$ can be represented (not necessarily uniquely) as:

\[
X_1 +_{\varepsilon} \mathcal{X} \text{rec}_{\mathcal{X}} ((X_1 := t_1; \ldots ; X_m := t_m))
\]

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

5 Transaction Language

Our view maintenance technique works in conjunction with a simple transaction language in the spirit
larly, we show in Section 7 that query decomposition, 
explicitly include . . .

Definition 6.1 Let \( Q \) be a query in UnQL. We say
that \( Q \) is decomposable if:

1. For all \( t, t' \), \( Q(t \uplus t') = Q(t) \uplus Q(t') \), and
2. For all \( t, t' \), \( Q(t; t') = (Q(t); Q(t')) \).

\( t' \) in item 1 and both \( t \) and \( t' \) in item 2 are trees with
named inputs, hence the definition only makes sense for
queries acting on databases with named inputs.

View maintenance for decomposable queries is easy: e.g. after an update \( DB' := DB \uplus \Delta \), we have to up-
delete the view \( V = Q(DB) \) to \( V' := V \uplus Q(\Delta) \). Similarly, we show in Section 7 that query decomposition,
in the sense of Section 1, is easily done for decomposable
queries. For this we need:

Proposition 6.2 If \( Q \) is decomposable then:

\[
Q(\text{rect} (X_1 := t_1; \ldots X_k := t_k)) = \\text{rect} (Q(X_1 := t_1), \ldots Q(X_k := t_k))
\]

Proof: We only give an informal argument. Since rec
is the same as the infinite unfolding \( t + t + \ldots \), we
have \( Q(\text{rect} t) = Q(t+t+t+\ldots) = Q(t) + Q(t) + \ldots =
\text{rec} Q(t) \). Next we apply item 2 of Definition 6.1.

As we shall see, not every UnQL query is decompos-
able. However, in the theorem below, we prove that
eyery query in the “positive, join-free fragment”
of UnQL (definition in Section 8) can be obtained from
a decomposable one with minor post-processing: this
is the main result of our paper, and we will sketch the
proof in Section 9.

Theorem 6.3 For any query \( Q \) in the positive, join-
free fragment of UnQL, there exists a (positive, join-
free) query \( Q' \) and constant tree \( \alpha \) such that the fol-
lowing conditions hold:

1. \( \alpha = \alpha \uplus Q' \).
2. \( Q' \) is decomposable.
3. The cost of evaluating \( Q' \) is no larger than that of
   a naive evaluation of \( Q \).

7 Applications

We show here how Theorem 6.3 can be applied to
query decomposition and view maintenance.

7.1 Query Decomposition

For illustration we shall assume that the database
is stored on two sites \( s \) and \( s' \): the technique de-
scribed here generalizes straightforwardly to an arbi-
trary number of sites. As in Section 4 we assume that
\( DB \) is given in the canonical form \( X_1 \uplus DB_0 \) where
\( DB_0 = \text{rec}_X (t; t') \) with \( t \) residing on site \( s \) and \( t' \) on
site \( s' \). Let \( Q_0 \) be the query \( Q_0(DB_0) \) in the
form:

\[ Q_0(DB_0) \equiv Q(X_1 \uplus DB_0) \] 3

We proceed as follows:

1. Apply Theorem 6.3 to decompose \( Q_0 \) into:
   \[ Q_0(DB_0) \equiv \alpha \uplus Q'(DB_0) \]. Hence we have to
compute \( Q'(DB_0) \), which is \( Q'(\text{rec}_X (t; t')) \).
2. By item 2 of Theorem 6.3 and Proposition 6.2,
   \[ Q'(\text{rec}_X (t; t')) = \text{rec}_{X'} (Q'(t); Q'(t')) \], where \( X' \)
is the set of inputs and outputs of \( (Q'(t); Q'(t')) \).
   Hence we compute \( Q'(t) \) at site \( s \), and indepen-
dently \( Q'(t') \) at site \( s' \).
3. Send \( Q'(t) \) and \( Q'(t') \) to the client site: this step
   is potentially inefficient and will be refined below.
The client now holds \( (Q'(t); Q'(t')) \).
4. The client computes \( \text{rec}_{X'} (Q'(t); Q'(t')) \) (which
   consists in drawing \( e \) edges from outputs to in-
puts), thus obtaining \( Q'(DB_0) \). Finally it com-
putes \( Q_0(DB_0) \), as shown in step 1.

\[ \text{Proposition 6.2 only applies to queries expecting databases}
with named inputs, hence it does apply to \( Q_0 \), but not to \( Q \)
directly. \]

Figure 11: (a) A relational database with markers. (b)
Same database after an update.

of [GL95], which we describe next. A transaction is a
sequence of atomic transactions of the form (1) \( DB :=
DB \uplus \gamma \Delta, \) for \( \gamma' \subseteq \gamma, \) or (2) \( DB := DB \, \text{replace}_{X', \Box} \). We have defined \( \uplus \)
earlier. For the replace operation, assume \( DB \) is given in the canonical form (equation 1), and \( X' = (X_k, X_{k+1}, \ldots, X_m, X_{m+1}, \ldots, X_p) \) (that is
\( X \cap X' = \{ X_k, \ldots, X_m \} \)). Then, for \( \Box \) a tree with \( X' \)
entries, say \( \Box = (X_k := \Box_k, X_{k+1} := \Box_{k+1}, \ldots, X_p :=
\Box_p) \), \( DB \, \text{replace}_{X', \Box} \) is defined to be:

\[
X_1 \uplus X', \text{rec}_{X'} \left( X_1 := t_1, \ldots X_{k-1} := t_{k-1}, X_k := \Box_k, \ldots, X_p := \Box_p \right)
\]

Note that replace is rather unorthodox, in that it is de-
fined on the particular representation of the database,
and not on its meaning. More precisely, one can find
trees \( DB \) and \( DB' \) which are bisimilar, hence equal
for our purposes, but for which \( DB \, \text{replace} \Box \) is not
bisimilar to \( DB' \, \text{replace} \Box \).

Example 5.1 Consider the relational database \( DB \)
in Figure 6. To model insertions in the relations \( r_1 \) and \( r_2 \), we redesign \( DB \) by introducing two markers
\( X_1 \) and \( X_2 \), as in Figure 11. Then a traditional insertion
operation \( \text{DB} := \text{DB} \, \text{replace} X \), would be:

\[
\text{DB} := \text{DB} \uplus X_1 \left( \text{X_1 := (tup := \{ m \Rightarrow b, n \Rightarrow a \})} \cup X_1 \right)
\]

The result is shown in Figure 11. Note that we ex-
plicitly include \( \ldots \cup X_1 \), in order to allow for future
insertions.

6 Main Result

Our key technique for query decomposition and view
maintenance is to transform queries into decomposable
queries.

Definition 6.1 Let \( Q \) be a query in UnQL. We say
that \( Q \) is decomposable if:

1. For all \( t, t' \), \( Q(t \uplus t') = Q(t) \uplus Q(t') \), and
2. For all \( t, t' \), \( Q(t; t') = (Q(t); Q(t')) \).

We proceed as follows:

1. Apply Theorem 6.3 to decompose \( Q_0 \) into:
   \[ Q_0(DB_0) \equiv \alpha \uplus Q'(DB_0) \]. Hence we have to
compute \( Q'(DB_0) \), which is \( Q'(\text{rec}_X (t; t')) \).
2. By item 2 of Theorem 6.3 and Proposition 6.2,
   \[ Q'(\text{rec}_X (t; t')) = \text{rec}_{X'} (Q'(t); Q'(t')) \], where \( X' \)
is the set of inputs and outputs of \( (Q'(t); Q'(t')) \).
   Hence we compute \( Q'(t) \) at site \( s \), and indepen-
dently \( Q'(t') \) at site \( s' \).
3. Send \( Q'(t) \) and \( Q'(t') \) to the client site: this step
   is potentially inefficient and will be refined below.
The client now holds \( (Q'(t); Q'(t')) \).
4. The client computes \( \text{rec}_{X'} (Q'(t); Q'(t')) \) (which
   consists in drawing \( e \) edges from outputs to in-
puts), thus obtaining \( Q'(DB_0) \). Finally it com-
putes \( Q_0(DB_0) \), as shown in step 1.

\[ \text{Theorem 6.3 only applies to queries expecting databases}
with named inputs, hence it does apply to \( Q_0 \), but not to \( Q \)
directly. } \]

233
Parts of the graphs $Q'(t)$ and $Q'(t')$ sent in step 3 may turn out (during step 4) to be inaccessible from the root of the final result $Q_0(DB_0)$. To avoid sending these useless fragments in step 3, we refine it as follows:

- Compute the accessibility graph $G$ for $(Q'(t); Q'(t'))$: nodes are markers $X'$, edges are pairs $(X, X')$ s.t. $X'$ is accessible from input $X \in X'$ in $(Q'(t); Q'(t'))$. The two fragments of $G$ are computed independently on sites $s$ and $s'$, then sent to the client.

- The client computes the transitive closure of $G$: this gives the accessibility graph for $Q'(DB_0) = \text{rec}_X' (Q'(t); Q'(t'))$. Using this and the tree $\alpha$, the client computes the set $X'_0 \subseteq X'$ of markers actually used in the concatenation $Q_0(DB_0) = \alpha + x Q'(DB_0)$, i.e. $\alpha + x Q'(DB_0)$ is equal to $\alpha + x'_0 Q'(DB_0)$.

- The client sends the set $X'_0$ to $\alpha$ and $s'$. Next, step 3 is resumed, but having each server sent only the “accessible” fragment of $Q'(t)$ (and $Q'(t')$ respectively) to the client, i.e. that under the inputs in $X'_0$.

Finally we argue that the total cost of evaluating our decomposed query is no larger than that of computing $Q(DB)$ on a centralized version of $DB$, using a naive evaluation method. Indeed $Q$ and $Q'$ are equally expensive to compute. The key observation is that, under a non-optimized evaluation, $Q'(\text{DB})$ consists of the following. Let $a$ be a view

$$\text{vext}_a (Q_1)(Q_2) \triangleq \text{vext}_a (Q_1 ; Q_2) = \text{vext}_a (Q_1 + Q_2)$$

for (Q(t); Q(t')); nodes are markers $X'$, edges are pairs $(X, X')$ s.t. $X'$ is accessible from input $X \in X'$ in $(Q(t); Q(t'))$. The two fragments of $G$ are computed independently on sites $s$ and $s'$, then sent to the client.

Theorem 6.3. [BDHS96a] introduces the language UnQL as a query language for browsing unstructured data, or data whose structure is only partially known. It is shown that UnQL is equivalent to a calculus, called UnCAL, much in the same way in which a certain fragment of SQL is equivalent to the relational algebra. Here we briefly review UnCAL. This is a rather dry formalism, and we refer the reader to [BDHS96a] for a presentation of the friendlier, and equivalent, UnQL language.

Figure 12 lists the constructs of a monotone fragment of UnCAL, which we call UnCAL. UnCAL is obtained by restricting UnCAL in two ways: (1) the emptiness test $\text{isEmpty?}(t)$ is removed, and (2) the $\text{vext}$ construct is replaced with the more restrictive $\text{vext}$ one. One can show that all queries expressed in UnCAL are “monotone”, in a way which can be made formal [Bun95].

We briefly describe the constructs in Figure 12, where $Q_1$, $Q_2$ are queries, $a, a'$ are label variables or constants, $p$ is a user-defined predicate. $DB$ stands for the input database, $\{\}$ returns the empty set (one node, no edges). $a \Rightarrow Q$ returns a singleton tree (here $a$ is a label constant or a label variable), while $Q_1 \cup Q_2$ returns the union of two trees. We can represent the latter conveniently using $\varepsilon$ edges, as in Figure 12. $Q_1 \text{++}_A Q_2$ concatenates $Q_1$ with $Q_2$ (Section 3).

The most complex construct is $\text{vext}_A (\lambda x . Q_1)(Q_2)$. Here $Q_1$ is an expression denoting a tree with inputs $A$ and outputs $\Delta$, which may have $x$ as a free label variable. Then $\text{vext}_A (\lambda x . Q_1)(Q_2)$ returns a tree obtained from $Q_2$ by replacing every edge labeled $a$ with the tree $Q_1[a/x]$. Figure 12 tries to illustrate this action for the case when $\Delta = \emptyset$ has a single marker. We will illustrate $\text{vext}$ below.

Next, $A := Q$ renames the unique, anonymous input of $Q$ to $A$: i.e. when $Q \in \text{Tree}_x$, then $A := Q$ is in $\text{Tree}_x(Q)$. In the juxtaposition $(Q_1; Q_2)$, the inputs of $Q_1$ and $Q_2$ have to be disjoint: the result will be the union of the two graphs (which is not $Q_1 \cup Q_2$). That is, when $Q_1 \in \text{Tree}_{x_1}$, $Q_2 \in \text{Tree}_{x_2}$ and $x_1 \cap x_2 = \emptyset$, then $(Q_1; Q_2)$ is in $\text{Tree}_{x_1 \cup x_2}$. We abbreviate $(Q_1; (Q_2; (\ldots; (Q_n; \ldots)))$ with $(Q_1; \ldots; Q_k)$. $\{\}$ is the null tree (no nodes, no edges). We have $(Q_1; Q_2) = (Q_2; Q_1)$ and $(Q; \{\}) = Q$. $A$ stands for a marker.

A query is an expression with no free variables, except for $DB$, which is the query input. A query may use markers of its own, in constructs like $\text{++}_A, \text{vext}_A, A := Q, A$, which we denote with $A, A', A_1, A_2, \ldots$, and which are distinct from the markers $X, X_1, \ldots, Y, Y_1, \ldots$ in the database $DB$. This
is much like in the case of oid's: $X, X_1, \ldots$ are like external oid's, and the query $Q$ cannot access them. We allow one exception to this rule: a query on a database with named inputs may use the unique marker $X_1$ denoting the "root" input to DB: thus, the query $Q(DB) \overset{def}{=} X_1 + X_1 DB$ is legal, but $X_2 + X_1 DB$ or $DB + X_1 (Y_1 := \{a\})$ are not.

The main construct of the language is $\text{vext}$. First we will explain how $\text{vext}_A(\lambda x. Q)(Q')$ interacts with the input and output markers of $Q'$. Recall that $Q$ must have $A = \{A_1, \ldots, A_p\}$ as inputs and outputs, and assume that $Q'$ has inputs $X = \{X_1, \ldots, X_m\}$ and outputs $Y = \{Y_1, \ldots, Y_n\}$. By definition $\text{vext}$ returns a tree with $p \cdot m$ inputs and $p \cdot n$ outputs. To name them, we denote with $U \cdot V$ a new, distinct marker, for any two markers $U, V$. We require this operation to be injective, i.e. $U \cdot V = U' \cdot V' \Rightarrow U = U'$ and $V = V'$. Then, by definition, the inputs of $\text{vext}_A(\lambda x. Q)(Q')$ are $A \cdot X \overset{def}{=} \{A_1 \cdot X_j | A_i \in A, X_j \in X\}$, and the outputs are $A \cdot Y$. By convention, when $Q'$ has a single anonymous input, then $\text{vext}_A(\lambda x. Q)(Q')$ has inputs $A$. We illustrate next $\text{vext}$ with a detailed example, and refer to [BDHSS96a] for a general definition of $\text{vext}$.

Example 8.1 Recall the query in Example 1.1 which returns all papers in the Computer Science Department. We consider here a variant, which stops following a path once it finds a "Papers" edge before a "CS-Department": intuitively this is an optimization, if one assumes that no department edge ever occurs after a "Papers" edge. Then $Q$ can be expressed as follows: $A_1 ++ (\text{vext}_A(\lambda x. Q_1(x))(DB))$,

where $A = \{A_1, A_2, A_3\}$ and $Q_1(x)$ is:

- if $x = "CS-Department"$ then
  $(A_1 := A_2; A_2 := A_3; A_3 := \{x \Rightarrow A_3\})$
- else if $x = "Papers"$ then
  $(A_1 := \{\}; A_2 := \{x \Rightarrow A_3\}; A_3 := \{x \Rightarrow A_3\})$
- else $(A_1 := A_2; A_2 := A_3; A_3 := \{x \Rightarrow A_3\})$

$Q_1(x)$ is best visualized graphically, as in Figure 13 (a). Figure 13 (b) contains an example of a database with three outputs $X = \{X, Y, Z\}$. Then $\text{vext}_A(\lambda x. Q_1)(DB)$ returns a database with 9 outputs $A \cdot X$, which is represented in Figure 13 (c). After eliminating $\varepsilon$ edges and the unaccessible part of the output graph we obtain the database in Figure 13 (d). Note that only the outputs $A_2 \cdot Y$ and $A_2 \cdot Z$ remain in the result. The intuition is that the markers $X, Y, Z$ are "copied" in the output, and "tagged" with the region where they were found, as described in Section 2: that is $A_2 \cdot Y$ means that $Y$ has been found in region 3, while $A_2 \cdot Z$ means that $Z$ was found in region 2. Note that marker $X$ disappears: no matter how we extend $DB$ at marker $X$ (e.g. an update, or a link to another site), $Q(DB)$ will not be affected.

$Q$ is not decomposable in the sense of Definition 6.1. However the sub-query $Q'(t) = \text{vext}_A(\lambda x. Q_1(t))(t)$ is decomposable: when $t \in \text{Tree}_X$, $t' \in \text{Tree}_Y$ then $Q'(t) \in \text{Tree}_A$, $Q'(t') \in \text{Tree}_A$ and $Q'(t ++ t') =$

The original query in Example 1.1 would have had $A_1 := A_1$ instead of $A_1 := \{\}$ in the case $x = "Papers"$. 

---

![Figure 12: The UnCALe calculus.](image-url)
The **query evaluation** method for UnCAL consists in manipulating labeled graphs, as suggested in Figure 12, and occasionally “cleaning up” the graph, by reducing it under bisimilarity (this also eliminates the unaccessible parts). For the main construct, \( \text{vext} \), there are two evaluation strategies, which we illustrate on \( \text{vext}(\pi x.\query(DB)). \) (1) Start from the root in \( \text{DB} \) and traverse it recursively, using \( \query \) as an automata. For the example in Figure 13 (b), this will produce the directly the graph in (d), without ever touching the unaccessible parts. (2) Process each edge of \( \text{DB} \) independently, by replacing it with \( \query \), a process illustrated in Figure 13 (c): here a clean-up phase is required to get to (d). Method (1) can be more efficient, and we called the “optimized” evaluation; it needs special care to deal with cycles; method (2) makes more sense on databases with more than one input, since there is no root to start from, but is more “naive” because it may end up constructing unnecessary fragments of a graph, which are later eliminated. The statement in item 3 of Theorem 6.3 holds for both methods. However in the query decomposition method recall that we also need to replace \( \query(\text{rec}(t, t')) \) with \( \text{rec}(\query(t), \query(t')) \) for some decomposable \( \query \): the cost of evaluating both of them is the same under method (2), but method (1) could be more efficient on \( \query(\text{rec}(t, t')) \), because it may avoid accessing parts of the database which are not needed. Therefore our query decomposition method has the same cost as computing \( \query(\text{DB}) \) on a centralized version of \( \text{DB} \) using method (2), but may be more expensive than \( \query(\text{DB}) \), when method (1) is used.

We call a UnCAL expression \( \query \) **constant** if it doesn’t mention the input database \( \text{DB} \). Note that in the example above, the sub-query \( \query_1 \) in \( \text{vext}_A(\pi x.\query_1)(\text{DB}) \) is a constant expression. We call a query \( \query \) **join-free** if in every sub-query of the form \( \text{vext}_A(\pi x.\query_1)(\text{DB}) \), (1) \( \query_1 \) is constant and (2) \( \query_1 \) does not use any additional output markers besides \( A \). It is easy to extend the results in [BDHS96a] to show that the relational algebra operations union, selection and projection over relational databases expressed as trees, can be expressed as join-free UnCAL queries, and that intersection, cartesian product, and eqjoin can be expressed in UnCAL (but not as join-free queries). Condition (2) is imposed because we want to disallow self concatenation, like in \( \text{DB} \looparrowright \text{DB} \), in order for Theorem 6.3 to hold. Direct self concatenation is not possible in UnCAL, because we cannot use \( \text{DB} \) ’s output markers in a query. But without rule (2) we can still do a self-concatenation in the form \( \text{vext}_A(\pi x.\query_1)(\text{DB}) \looparrowright \text{DB} \) in which \( \query_1 \) uses the marker \( B \not\in A \) to place it deep into \( \text{DB} \) and leave it there.

Finally, for the purpose of the results in Section 6, we call an UnQL query \( \query \) **positive** if its translation into UnCAL, given in [BDHS96a] is in UnCAL (i.e. doesn’t use isEmpty?). Similarly, we call \( \query \) **join-free** iff its translation is join-free. It is not difficult to design syntactic conditions which are sufficient for UnQL queries to be positive and/or join-free.

### 9 Decomposition Rules

We sketch here the proof of Theorem 6.3, by describing the decomposition rules which, when applied to some query \( \query \), transform it into \( \alpha \looparrowright \query' \). The rules are given in Figure 14, and apply inductively on the sub-queries of \( \query \). Note that in \( \text{vext}_A(\pi x.\query_1)(\text{DB}) \), \( \query_1 \) is a constant expression and hence does not need to be transformed. As a consequence the inductive transformation rules never reach an if – then – else construct. Before applying the rules, we transform the query \( \query \) ensuring that distinct \( \text{vext}_A \) subexpressions in \( \query \) use disjoint sets of markers \( A \): if not, then we simply rename the markers in some of the \( \text{vext}_A \) constructs.

For the base case, when \( \query = \text{DB} \in \text{Tkee}_A \), \( \alpha \) has to be taken as the “identity on the inputs \( X \),”, i.e. \( (X_1 := X_1; \ldots; X_m := X_m) \). This is not expressible in UnCAL, because we do not have access to the sets of markers \( X \), so we add a new construct, \( \text{idx} \), denoting the identity on the inputs of \( \text{DB} \). Hence, strictly speaking \( \alpha \) will be an expression in UnCAL + \( \text{idx} \).

First we explain the juxtapositions in Figure 14. Consider any two sub-queries \( \query_1 \) and \( \query_2 \) of some larger query \( \query \), and let \( \query_1' \) and \( \query_2' \) be their transformations. \( \query_1' \) and \( \query_2' \) may have common input markers, hence the \( (\query_1'; \query_2') \) construct is not quite correct. However we observe that the only possible common inputs are those in \( \text{DB} \). More precisely, for \( \query_1' \) and \( \query_2' \) of the following two cases arise: (1) every input in \( \query_1' \) is of the form \( A \cdot (\ldots) \), i.e. tagged with some marker \( A \) used in a \( \text{gext}_A \) subexpression of \( \query_1 \), hence private to \( \query_1 \); in this case all inputs in \( \query_1' \) are disjoint from those in \( \query_2' \) and the juxtaposition \( (\query_1'; \query_2') \) is correct. (2) \( \query_1' = (DB; \query''_1) \), with all inputs of \( \query''_1 \) tagged with private marker of \( \query_1 \). In this case we inspect \( \query_2' \); if it also has the form \( (DB; \query''_2) \), then we adopt \( (\query_1''; \query_2'' \) the meaning \( DB; \query_1''; \query_2'' \), i.e. copy only once the common part which is \( DB \).

We illustrate the correctness of the rules in Figure 14 for \( \text{vext} \), \( \cup \) and \( \looparrowright \). Consider the query \( \query = \text{vext}_A(\pi x.\query_1)(\text{DB}) \). First apply induction hypothesis to get \( \query_2 = \alpha_2 \cup \query_2' \) (recall that \( \query_1 \) is constant).

Then Figure 14 defines \( \query' \triangleq \text{vext}_A(\pi x.\query_1)(\text{DB}) \) and \( \alpha \triangleq \text{vext}_A(\pi x.\query_1)(\alpha_2) \). We will use the following two equations:

\[
\text{vext}_A(\pi x.\query_1)(t_1 \looparrowright t_2) = \text{vext}_A(\pi x.\query_1)(t_1) \looparrowright \text{vext}_A(\pi x.\query_1)(t_2) \quad (2)
\]

\[
\text{vext}_A(\pi x.\query_1)(t_1; t_2) = (\text{vext}_A(\pi x.\query_1)(t_1); \text{vext}_A(\pi x.\query_1)(t_2)) \quad (3)
\]

Equation 2 is from [BDHS96a], while 3 is a direct consequence of the definition of \( \text{vext} \). First we check that \( \query = \alpha \cup \query' \), which follows from equation 2. Next, using both equations 2 and 3, and the induction hypothesis that \( \query_2' \) is decomposable, one can easily check that \( \query' \) is decomposable too.

Consider now the case \( \query = \query_1 \cup \query_2 \). First apply induction hypothesis to get \( \query_i = \alpha_i \cup \query_i' \), \( i = 1,2 \).
Figure 13: Illustration of $\text{vext}(\lambda x. Q)(DB)$.

Then Figure 14 defines $Q' \equiv (Q'_1 ; Q'_2)$, and $\alpha \equiv \alpha_1 \cup \alpha_2$. First check item 1 of Theorem 6.3:

\[
\alpha + Q' = (\alpha_1 \cup \alpha_2) + (Q'_1 ; Q'_2) = (\alpha_1 + Q'_1) \cup (\alpha_2 + Q'_2) = Q_1 \cup Q_2
\]

The first equality is true because of the separation of the inputs markers for $Q'_1$ and $Q'_2$, as explained above. They may share only input markers from the inputs of $DB$, and in this case their common part is exactly $DB$: hence $(\alpha_1 \cup \alpha_2) + (Q'_1 ; Q'_2) = (\alpha_1 + Q'_1) \cup (\alpha_2 + Q'_2)$.

Next we check item 2:

\[
Q'(t_1 \uparrow t_2) = (Q'_1(t_1 \uparrow t_2); Q'_2(t_1 \uparrow t_2)) = (Q'_1(t_1); Q'_2(t_1)) + (Q'_1(t_2); Q'_2(t_2)) = Q'_1(t_1) \uparrow Q'_2(t_2)
\]

For the $Q = Q_1 + \alpha_1 Q_2$ construct, we again apply induction first, $Q_i = \alpha_i + Q'_i$, $i = 1, 2$. Obviously $(Q'_1 ; Q'_2)$ is decomposable (as above), so it remains to show that $Q_1 \uparrow + \alpha_1 Q_2 = (\alpha_1 + \alpha_2) + (Q'_1 ; Q'_2)$. For this, we first show by induction that every output marker of $Q'_1$ is "tagged" with some output marker in $\mathcal{Y}$. Hence none of markers in $\mathcal{A}$ may appear in $Q'_1$ (this is a consequence of condition (2) of the join-freeness definition of Section 8). Hence we have:

\[
Q_1 \uparrow + \alpha_1 Q_2 = (\alpha_1 + Q'_1) \uparrow + \alpha_2 (\alpha_2 + Q'_2) = (\alpha_1 + \alpha_2) + (Q'_1 ; Q'_2)
\]

10 Conclusions

We have described a query decomposition method which can be used to compute efficiently queries.
Figure 14: Rules for transforming any UnCALe join-free query $Q$ into $\alpha + Q'$, with $Q'$ decomposable.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$Q'$</th>
<th>$Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$id_{Q'}$</td>
<td>$DB$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>${a \rightarrow Q_1}$</td>
<td>$\emptyset$</td>
<td>${a \rightarrow \alpha_1}$</td>
</tr>
<tr>
<td>$Q_1 \cup Q_2$</td>
<td>$Q'_1$, $Q'_2$</td>
<td>$\alpha_1 \cup \alpha_2$</td>
</tr>
<tr>
<td>$Q_1 ++_A Q_2$</td>
<td>$Q'_1$, $Q'_2$</td>
<td>$\alpha_1 ++_A \alpha_2$</td>
</tr>
<tr>
<td>$\text{vext}_A(\lambda z. Q_1), (Q_2)$</td>
<td>$\text{vext}_A(\lambda z. Q_1), (Q_2)$</td>
<td>$\text{vext}_A(\lambda z. Q_1), (Q_2)$</td>
</tr>
<tr>
<td>if $C$ then $Q_1$ else $Q_2$</td>
<td>$N/A$</td>
<td>$A := \alpha_1$</td>
</tr>
<tr>
<td>$A := Q'_1$</td>
<td>$Q'_1$</td>
<td>$A := \alpha_1$</td>
</tr>
<tr>
<td>$(Q_1; Q_2)$</td>
<td>$(Q'_1; Q'_2)$</td>
<td>$(\alpha_1; \alpha_2)$</td>
</tr>
<tr>
<td>$A$</td>
<td>$A$</td>
<td>$A$</td>
</tr>
</tbody>
</table>

in unstructured query languages on distributed data sources. The decomposition is "efficient" in the sense that, under a certain naive evaluation strategy, the decomposed query is no more expensive to compute on the distributed database, than were the original if applied to a centralized database. Also, we have proposed two update operations for the underlying data model, and shown that the same query decomposition method can be used to derive a simple view maintenance method.

Our methods work without any knowledge about the structure of the database. While sometimes this can be useful, in general it is more a limitation than a virtue: some simple knowledge about the database could improve the query decomposition. In future work we plan to develop techniques to incorporate such information in the query decomposition.

Acknowledgments I would like to thank Peter Buneman and Gerd Hillebrand for their comments.

References


