Complete Answer Aggregates for Structured Document Retrieval

Holger Meuss Klaus U. Schulz

Center for Information and Language Processing (CIS)

University of Munich

Oettingenstr. 67 80538 Munich, Germany

e-mail: {meuss,schulz}@cis.uni-muenchen.de

May 3, 1999

Abstract

The use of markup languages like SGML, HTML, or XML for encoding the structure of documents has lead to many databases where entries are adequately described as trees. In this context querying formalisms are interesting that offer the possibility to refer both to textual content and logical structure. If answers are formalized as mappings to–or subtrees of–the database, a simple enumeration of all answers will often suffer from the effect that many mappings/subtrees have common subparts. From a theoretical point of view this may lead to an exponential time complexity of the presentation of all answers. Concentrating on the language of so-called tree queries—a variant and extension of Kilpeläinen's Tree Matching formalism—we introduce the notion of a "complete answer aggregate" for a given query. A complete answer aggregate offers a compact view of the set of all answers that seems attractive for practical use in IR systems. Since complete answer aggregates use an exhaustive structure sharing mechanism their maximal size is of order $\mathcal{O}(d \cdot h \cdot q)$ where d (q) is the size of the database (query) and h is the maximal depth of a path of the database. We give an algorithm that computes a complete answer aggregate for a given tree query in time $\mathcal{O}(d \cdot log(d) \cdot h \cdot q)$. For the sublanguage of so-called rigid tree queries, as well as for so-called "non-recursive" databases, an improved bound of $\mathcal{O}(d \cdot log(d) \cdot q)$ is obtained. The algorithm is based on a specific index structure that supports practical efficiency.

Keywords: Tree databases, tree matching, information retrieval, structured documents, query languages, logic.

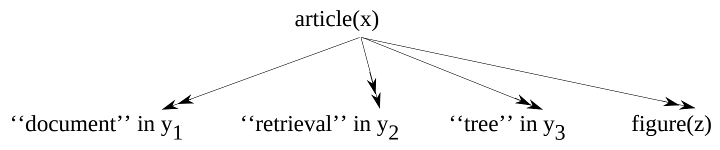
1 Introduction

Databases where entries are described as trees are interesting for many reasons. Most importantly, the use of markup languages like HTML ([W3C98c]) or XML ([W3C98b]) for documents in the World Wide Web, as well as the use of other SGML ([ISO86, Gol90]) dialects for document exchange in enterprises and institutions has lead to huge repositories where both logical structure and textual contents of documents are explicitly represented using trees of a particular form. The interest in conceiving these document collections as databases has been stressed by various authors ([FLM98, Suc98b]) and led to a special W3C workshop dedicated to XML query languages ([W3C98a]). Tree databases play also an important role in the area of computational linguistics. Here during the last years large-scale databases with parse trees of sentences and phrases have been built up (e.g., [MSM93, OMM98]), both for theoretical studies and for practical use in systems, e.g. for machine translation. From a more basic point of view, trees undoubtfully represent a very natural organization scheme, and it seems realistic to expect a growing number of applications of tree database techniques in the middle term.

In the field of Information Retrieval (IR) various models and query languages have been proposed in the meantime that take both logical structure and the contents of documents into account (e.g. [GT87, Bur92, KM93, NBY97, MAG$^+$97]), each suggesting yet another compromise between expressiveness and efficiency (see [BYN96] and [Loe94] for surveys). One of the most expressive query languages is Kilpeläinen's Tree Matching formalism [Kil92]. In this approach, partial descriptions of trees are used as queries, and answers are formalized as homomorphic embeddings that correspond to subtrees of the database.

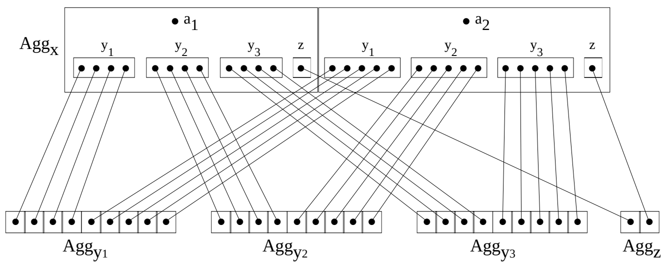
In this paper we introduce a formalism that can be considered as a generalization of Tree Matching. The "tree queries" of our query language essentially add to Kilpeläinen's matching expressions some additional flexibility and expressivity. The main contribution of this paper, though, is a new concept for computing and presenting answers. This concept is not only relevant for the Tree Matching formalism but for each retrieval model where answers itself are structured and do not just return one single pointer to some offset point in a relevant document.

As an illustration, imagine a person that wants to retrieve articles that contain the key words "document", "retrieval", "tree" and have a figure. In the Tree Matching formalism – slightly modified here – this request can be formalized using a tree pattern of the form



Under any answer, node x (resp. z) has to be mapped to a document node with label "article" (resp. "figure"). Nodes y_1, y_2 and y_3 have to be mapped to document nodes with textual contents; the keywords that are specified in the pattern have to occur in the flat text dominated by these nodes. Double arrows indicate descendant (not necessary children) relationship. Since answers return the images of all the nodes, they may be used, for example, to collect all figures of articles that are conform with this description. Now imagine a database with two articles a_1 and a_2 that have the desired form. Article a_1 (resp. a_2) has four (five) paragraphs (paragraphs are assumed to have not any inherent logical structure and informally correspond to textual nodes), and both articles contain one figure. In both cases, each paragraph contains the three words "document", "retrieval", "tree". In this situation each of the nodes y_i can be mapped to each of the paragraphs. Since answers correspond to embedding homomorphisms, there is a total of $4^3 + 5^3 = 192$ possible answers to the query. If answers are presented via enumeration, the user will hardly have the patience to inspect all of them in order to find that there are just 2 articles and 2 figures participating to all these answers. Obviously, each formalism that offers a comparable expressivity and formalizes answers as mappings suffers from the same problem as long as answers are enumerated.

What we suggest instead is computation of a so-called *complete answer aggregate*, an abstract, intensional representation of the set of all answers. The structure of such an aggregate is derived from the query. Basically, each query node is used as a "container" that collects all the document nodes that represent a possible image ("target candidate") of the query node. In addition, each target candidate in a complete answer aggregate is enriched with some administrational information on possible descendants. For the given query and database the aggregate has the following *internal* representation.

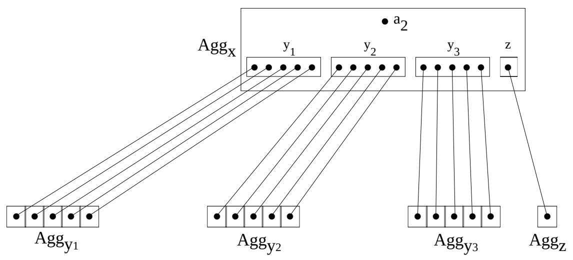


The record Agg_x collects the possible target candidates for the article node x. In our example this record has two fields, indicated by large rectangles. The first (second) field contains a pointer to article a_1 (resp. a_2) and a list of four arrays, corresponding to the four children of x in the query (small rectangles). These arrays are used to point to the fields of possible target candidates for the children, subject to the choice of $x = a_1$ (resp. $x = a_2$). There are four additional records, for y_1, y_2, y_3, and z respectively. Since these variables represent leaves of the query the structure of their answer records is simpler. Each record Agg_{y_i} just contains a list of pointers (only indicated by dots) to the $4 + 5 = 9$ paragraphs which represent the possible target candidates for the textual nodes y_i $(1 \leq i \leq 3)$. Eventually record Agg_z has two pointers, each representing a link to a figure.

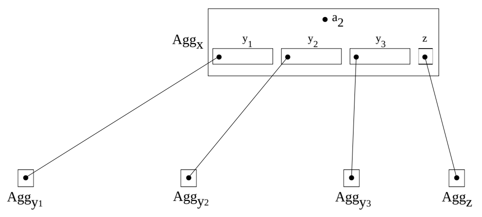
Externally, the aggregate can be presented to the user in many different ways. We may, e.g., just present the number of possible nodes of each record in a first step:

$$\begin{tikzcd}
& \text{article(x): }\mathbf{2} \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\text{``document'' in } y_1\text{: }\mathbf{9} & \text{``retrieval'' in } y_2\text{: }\mathbf{9} & \text{``tree'' in } y_3\text{: }\mathbf{9} & \text{figure(z): }\mathbf{2}
\end{tikzcd}$$

If the user wishes to inspect the two target candidates for the article node x more closely, the next step could be to show (in an appropriate way) the list with the articles a_1 and a_2 from Agg_x. If the reader then decides to inspect a_2, an internal view might temporally restrict the aggregate in the following way (*narrowing*).

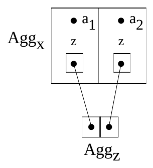


We may provide the user with the information that given his selection of a_1 there are now five possible target candidates for the each textual node, and a unique figure. The user might now decide to inspect the figure. If he also activates the first possible candidate for y_1, y_2 and y_3 respectively, the actual internal view

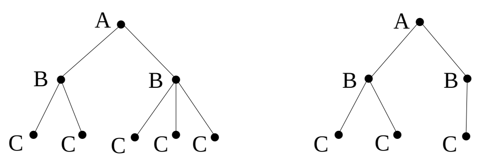


represents one of the 192 possible answers. In another situation the user might find that this particular answer is useless and reactivate the previous state.

Orthogonal to the construction of views by narrowing, the user may specify parts of the query that he does not want to inspect (still, these query parts may be meaningful since they trigger the selection of answers). In this case some of the query variables will not be represented in the answer aggregate. The specification of records Agg_x that are to be suppressed can be integrated in the query formulation or it can be part of an interactive process during inspection of answers. In the above example the user could for example suppress textual nodes, which would result in the following (internal) form of the aggregate



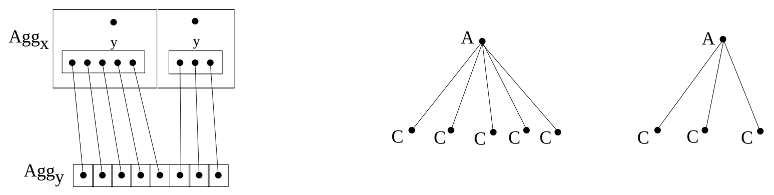
Answer aggregates can also be useful for dynamic change of the database. As an illustration, take a database with trees



The aim is to compute a simplified representation where nodes with label B are suppressed. This type of "structural simplification" is one of the operations discussed in [GT87]). The complete answer aggregate for the query

$$\begin{tikzcd}
A(x) \arrow[d] \\
C(y)
\end{tikzcd}$$

depicted below at the left-hand side, can immediately be translated into the required simplified view of the database on the right-hand side.



Yet another aspect where aggregates possibly can offer some new contribution is ranking of answers. In our first example we may use the information that a_2 has more relevant paragraphs than a_1 as a justification for an inverted enumeration $a_2 \to a_1$ of relevant articles. More sophisticated IR techniques for assigning weights to key words can perhaps be lifted to account for structure, assigning now weights to target candidates, depending on the number and weight of possible target candidates for the children.

These scenarios just indicate some possibilities for using the information that we obtain from an aggregate such as the one depicted above. We stop the discussion at this point, since it is not our principal concern to give an exhaustive list of all possible ways for using a complete answer aggregate. Instead, the main aim of the paper is to give a rigorous definition of this notion and a characterization of its mathematical and computational properties. The concept will be introduced in the context of a logical reformulation and extension of Kilpeläinen's Tree Matching formalism [Kil92].

The paper has the following structure. After some formal preliminaries in Section 2 we first formalize collections of tree-structured documents as so-called *relational document structures* in Section 3. A relational document structure is a conventional structure of first-order predicate logic that represents a document database. It should be stressed that this type of formalization can be used for arbitrary databases where entities represent trees. In this sense, databases with documents just serve as an illustrating example.

The query language is introduced in Section 4. In principle, the full first-order language associated with a given relational document structure can be considered as a query language. Under computational aspects, the sublanguage of *tree queries* seems to be of particular interest. For this latter language we give an algebraic reformulation of the query evaluation that underpins the direct relationship to the Tree Matching formalism.

Section 5 first introduces the concept of a *complete answer formula* for the subclass of so-called "simple tree queries". This notion yields a compact description of the full set of answers to a query in terms of a conventional formula of propositional logic. It is shown that for each simple tree query, Q, and each relational document structure, $\mathcal{D}$ there exists a unique complete answer formula that represents the set of all answers to Q in $\mathcal{D}$.

The "logical" notion of a complete answer formula is then translated into the "physical" notion of a *complete answer aggregate* for Q. We prove that the size of the complete answer aggregate for Q is bounded by $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$ where $|Q|$ is the size of Q, $|D|$ is the number of nodes of $\mathcal{D}$ and $h_{\mathcal{D}}$ is the maximal length of a path in $\mathcal{D}$. In a tree query we may refer to arbitrary descendants of a node. For the subclass of "rigid" queries, where we can only refer to immediate children of a node in a query, we give a better bound $\mathcal{O}(|Q| \cdot |D|)$. The improved bound is also obtained for "non-recursive" databases, i.e., databases where the same label does not occur twice in a path. Here we have to assume that each query node carries either some textual information or some label information. In a second step, all these notions are generalized to the class of so-called "partially ordered tree queries" where it is possible to impose restrictions on the left-to-right ordering of nodes in the query. The above bounds for the size of the complete answer aggregate of a query still hold for partially ordered queries.

In Section 6 we describe an algorithm that computes the complete answer aggregate for a given partially ordered tree query in time $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}} \cdot \log(|D|))$. For rigid queries, as well as for non-recursive databases, the algorithm runs in time $\mathcal{O}(|Q| \cdot |D| \cdot \log(|D|))$. In order to guarantee a good practical behaviour of the algorithm it is based on a special index structure.

We conclude in Section 7 with remarks on related work and on some open research issues. For the sake of readability some proofs are omitted in the running text and collected in an Appendix.

The present paper modifies and extends results of the first author presented in [Meu98].

2 Formal preliminaries

In this section we provide some basic mathematical background that is needed later. As usual, if R denotes a binary relation on a set M, then R^* (resp. R^+) denotes the reflexive-transitive (resp. transitive) closure of R.

Definition 2.1 A (finite, unordered) *tree* is a pair $(D,\rightharpoonup)$ where D is a finite, nonempty set and "$\rightharpoonup$" is a binary relation on D such that the following conditions are satisfied:

1. $\rightharpoonup$ is cycle-free, i.e., $\rightharpoonup^+$ is irreflexive,
2. for each $d' \in D$ there exists at most one element $d \in D$ such that $d \rightharpoonup d'$,
3. there exists exactly one element $d \in D$, called the *root* of $(D,\rightharpoonup)$ such that $d \rightharpoonup^* d'$ for each $d' \in D$.

The elements of D are called *nodes*. If $d \rightharpoonup d'$, then d' is a *child* of d, conversely d is called the *parent node* of d'. Condition 2) expresses that each node has at most one parent node. It follows from Condition 3) that a node having no parent node coincides with the root. Distinct children of a common parent node are called *siblings*. If $d \rightharpoonup^+ d'$, then d' is a *descendant* of d, conversely d is called an *ancestor* of d'. If $d \rightharpoonup^* d'$, then d' is a *reflexive descendant* of d. A node is a *leaf* if it does not have any child, otherwise it is an *inner node*.

A *path* of $(D,\rightharpoonup)$ is a sequence $\langle d_0, \ldots, d_n \rangle \in D^n$ where d_0 is the root of $(D,\rightharpoonup)$, d_n is a leaf of $(D,\rightharpoonup)$ and $d_i \rightharpoonup d_{i+1}$ for $0 \le i \le n-1$. A *partial path* of $(D,\rightharpoonup)$ is a prefix of a path of $(D,\rightharpoonup)$.

Definition 2.2 An *ordered tree* is a tree $(D,\rightharpoonup)$, together with a strict partial order $<^D_{lr}$ on D, called *left-to-right ordering*, that has the following properties:

1. $<^D_{lr}$ relates two nodes $d_1 \neq d_2$ (in the sense that either $d_1 <^D_{lr} d_2$ or $d_2 <^D_{lr} d_1$) iff neither d_1 is a descendant of d_2 in $(D,\rightharpoonup)$ nor vice versa,
2. if d_1 and d_2 are siblings, then $d_1 <^D_{lr} d_2$ implies that $d'_1 <^D_{lr} d'_2$ for all reflexive descendants d'_1 and d'_2 of d_1 and d_2 respectively.

If $(D,\rightharpoonup)$ is an ordered tree, the union of descendant relationship with left-to-right ordering is called the *pre-order relationship* and denoted $<^D_{pr}$. The following lemma is a trivial consequence of the definitions.

Lemma 2.3 *Let $(D, \rightharpoondown)$ be an ordered tree, let $d, e \in D$. Then $d <_{lr}^{D} e$ iff there exists a node $d' \in D$ such that $d <_{lr}^{D} d'$ and $d' \leq_{p}^{D} e$.*

3 Modeling documents as tree structures

As established by international standards like SGML [ISO86], we conceive documents as trees where nodes and edges are used to model the logical structure of the document. The leaves of a document tree contain the actual flat textual content of the document. Obviously, in order to represent databases with structured documents as trees we have to adapt the basic data structure. As a first step we introduce a formal distinction between "structural" nodes and "textual" nodes. Depending on the application area it might be interesting to model additional relations on the nodes, and to make these relations available in the querying process. One important example that we discuss below is the left-to-right ordering between nodes. Since we do not want to restrict the discussion to a specific set of relations, other relations that might be relevant, such as e.g. inequality, semantic comparisons, typing information, attribute values, similarity or proximity, are treated in a generic way as abstract constraints, leaving apart their precise nature. The resulting structures for modeling document databases will be treated as conventional structures of first-order predicate logic in the following sections.

Text nodes and structural nodes

Let Σ and Λ denote two fixed disjoint alphabets, called *text alphabet* and *markup alphabet* respectively. We assume that the textual content of a document is modeled by elements of Σ^* (i.e., by strings over Σ) and the structural markup (labels of structural nodes) is modeled by symbols in Λ.

Definition 3.1 A *document structure* is a tuple $\mathcal{D} = (D_S, D_T, \rightharpoondown, Lab)$ where

1. $(D_T \cup D_S, \rightharpoondown)$ is a tree where D_T and D_S represent disjoint sets of nodes,
2. Lab is a function that assigns to each node $d \in D_T$ a string $Lab(d) \in \Sigma^*$ and to each node $d \in D_S$ an element of Γ.

The following condition must hold:

3. Each node in D_T is a leaf of $(D_T \cup D_S, \rightharpoondown)$.

In the sequel, the nodes in D_T and D_S are called *text nodes* and *structural nodes* respectively, and $D := D_T \cup D_S$ will always denote the joint set of nodes.

With a *document path* of $\mathcal{D}$ we mean a path of $(D_T \cup D_S, \rightharpoonup)$. The document structure is called *non-recursive* if there does not exist any document path with two distinct structural nodes that are labeled with the same symbol in Γ.

It should be noted that we model each database as a single tree. Since we may always add to a given forest a new root element this decision does not impose any real restriction. Recall also that with a tree we mean an unordered tree if not said otherwise. Ordered trees can be modeled with the relations that we introduce now.

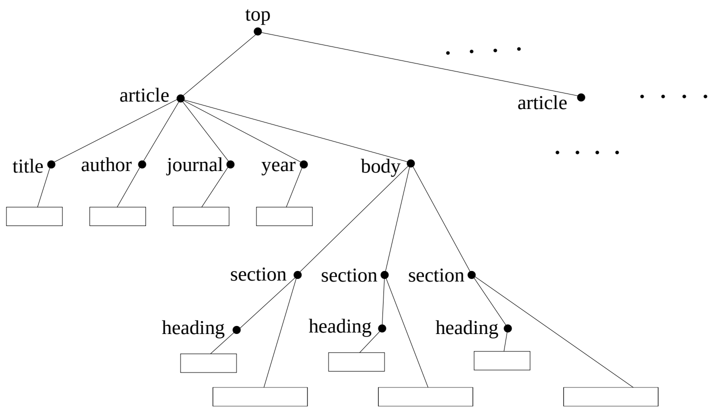
Ordering and other relations

Let $\mathcal{R}$ denote a fixed finite set of *relation symbols*, each equipped with a fixed arity.

Definition 3.2 A *relational document structure* is a tuple $\mathcal{D} = (D_S, D_T, \rightharpoonup, Lab, I)$, where $(D_S, D_T, \rightharpoonup, Lab)$ is a document structure and I is an interpretation function for $\mathcal{R}$, i.e., a mapping that assigns to every relation symbol $r \in \mathcal{R}$ of arity k a relation $I(r) \subseteq D^k$.

Definition 3.3 A relational document structure $\mathcal{D} = (D_S, D_T, \rightharpoonup, Lab, I)$ is *ordered* if $\mathcal{R}$ contains a symbol $<_{lr}$ where $(D_T \cup D_S, \rightharpoonup)$ together with $<_{lr}^{D} := I(<_{lr})$ is an ordered tree.

Example 3.4 The following figure depicts an ordered relational document structure that represents a collection of articles. Textual nodes are indicated by rectangles.



Remark 3.5 When using markup languages like SGML, the logical structure of

documents is encoded using a flat representation by means of delimiting tags. Various interesting aspects of the tree structure can be read off in a simple way:

1. two nodes d_1, d_2 stand in the pre-order relationship, $d_1 <^D_{pr} d_2$, iff the opening tag for d_1 precedes the opening tag for d_2,
2. two nodes d_1, d_2 stand in the left-to-right relationship, $d_1 <^D_{lr} d_2$, iff the closing tag for d_1 precedes the opening tag for d_2,
3. node d_2 is a descendant of node d_1 iff the opening tag for d_2 is between the two tags for d_1.

Following Definition 3.2, relational symbols in $\mathcal{R}$ are interpreted as relations over the set of nodes, D. Though this restriction simplifies the presentation it does not represent a principal limitation of the techniques suggested in this paper. In some examples we shall take a more liberal point of view and allow for relations over further sets. For example, attributes could be be modeled as relations between nodes, attribute names, and values.

4 A logical query language for relational document structures

In this section we define a logical query language $\mathcal{L}_\mathcal{R}$ for databases that are represented in the form of relational document structures. In principle the full first-order language associated with the given structure, presented in the first section, may be used as query language. For practical use in IR-systems, the sublanguage of so-called "tree-queries" seems to be of particular relevance. This sublanguage, introduced in the second section, will be studied in the following sections under various aspects.

4.1 The full first-order language

Recall that the alphabets Σ, Γ as well as the signature $\mathcal{R}$ are assumed to be fixed. In the sequel, let *Var* denote a countably infinite set of variables, denoted $x, y, z \ldots$, and let "*in*", "$\lhd$" and "$\lhd^+$" denote three new binary symbols.

Definition 4.1 The set of *atomic $\mathcal{L}_\mathcal{R}$-formulae* contains all formulae of the form

– $x \lhd y$, for $x, y \in Var$,

- $x \lhd^{+} y$, for $x, y \in \mathit{Var}$,
- $w \ \mathit{in} \ x$, for $x \in \mathit{Var}$ and $w \in \Sigma^{+}$,
- $M(x)$, for $x \in \mathit{Var}$ and $M \in \Gamma$,
- $r(x_1, \ldots, x_k)$ where $r \in \mathcal{R}$ has arity k and $x_1, \ldots, x_k \in \mathit{Var}$.

Atomic formulae of the form $r(x_1, \ldots, x_k)$ ($r \in \mathcal{R}$) are called *atomic constraints*. $\mathcal{L}_{\mathcal{R}}$*-formulae* are inductively defined as follows:

- each atomic $\mathcal{L}_{\mathcal{R}}$-formula is an $\mathcal{L}_{\mathcal{R}}$-formula,
- if φ and φ_1, φ_2 are $\mathcal{L}_{\mathcal{R}}$-formulae, then $\neg\varphi$, $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, and $(\varphi_1 \Rightarrow \varphi_2)$ are $\mathcal{L}_{\mathcal{R}}$-formulae,
- if φ is an $\mathcal{L}_{\mathcal{R}}$-formula and $x \in \mathit{Var}$, then $\exists x \varphi$ and $\forall x \varphi$ are $\mathcal{L}_{\mathcal{R}}$-formulae.

An $\mathcal{L}_{\mathcal{R}}$-formula φ is an $\mathcal{L}$*-formula* iff φ does not have a subformula that is an atomic constraint.

We write $x \lhd^{(+)} y$ when refering to formulae of the form $x \lhd y$ and $x \lhd^{+} y$ at the same time.

Let $\mathcal{D} = (D_S, D_T, \rightharpoonup, \mathit{Lab}, I)$ be a relational document structure. With a *variable assignment in* $\mathcal{D}$ we mean a (total) mapping $\nu : \mathit{Var} \rightarrow D$.

Definition 4.2 *Validity* of an atomic $\mathcal{L}_{\mathcal{R}}$-formula *in* $\mathcal{D}$ *under* ν is defined as follows[1]:

- $\mathcal{D} \models_{\nu} x \lhd y$ iff $\nu(x) \rightharpoonup \nu(y)$,
- $\mathcal{D} \models_{\nu} x \lhd^{+} y$ iff $\nu(x) \rightharpoonup^{+} \nu(y)$,
- $\mathcal{D} \models_{\nu} w \ \mathit{in} \ x$ iff $\nu(x)$ is a text node and $\mathit{Lab}(\nu(x))$ contains the word w,
- $\mathcal{D} \models_{\nu} M(x)$ iff $\nu(x)$ is a structural node and $\mathit{Lab}(\nu(x)) = M$,
- $\mathcal{D} \models_{\nu} r(x_1, \ldots, x_k)$ iff $\langle \nu(x_1), \ldots, \nu(x_k) \rangle \in r_{\mathcal{D}}$.

The validity relation is extended as usual to arbitrary $\mathcal{L}_{\mathcal{R}}$-formulae: Boolean connectives are just lifted to the meta-level, furthermore we define

- $\mathcal{D} \models_{\nu} \forall x \varphi$ iff for every node $d \in D$ we have $\mathcal{D} \models_{\nu'} \varphi$, where $\nu'(y) := \nu(y)$ for all variables $y \neq x$ and $\nu'(x) := d$,

[1]In the third condition we do not formally define what it means for a textual node to "contain" a given word w. This might mean "literal" containment, i.e., containment as a substring, or it might include linguistic normalization techniques like lemmatization or stemming.

- $\mathcal{D} \models_\nu \exists x\varphi$ iff there exists a node $d \in D$ such that $\mathcal{D} \models_{\nu'} \varphi$, where $\nu'(y) := \nu(y)$ for all variables $y \neq x$ and $\nu'(x) := d$.

An $\mathcal{L}_\mathcal{R}$-formula φ is *satisfiable* iff there exists a relational document structure $\mathcal{D}$ and an assignment ν such that $\mathcal{D} \models_\nu \varphi$. The set $\mathit{fr}(\varphi)$ of *free variables* of an $\mathcal{L}_\mathcal{R}$-formula φ is defined as usual. If $\mathit{fr}(\varphi)$ is given in a fixed order $\vec{x} = \langle x_1, \ldots, x_n \rangle$ and if $\vec{d} = \langle d_1, \ldots, d_n \rangle$ is a sequence of nodes of $\mathcal{D}$ we write $\mathcal{D} \models \varphi[d_1, \ldots, d_n]$ iff $\mathcal{D} \models_\nu \varphi$ for each variable assignment ν mapping x_i to d_i for $1 \leq i \leq n$.

Atomic formulae of the form "w *in* x" refer only to textual nodes. Obviously, it is desirable to refer in a similar way to the part of the flat text that is dominated by a structural node. If x is a variable of a query that contains a structural condition of the form $M(x)$, we may write $x \lhd^+ y \wedge w$ *in* y to express that the word w must occur in the text associated with node x. If more comfort is needed we might add some "syntactic sugar" and introduce a new type of atomic formula that represents the above conjunction.

Definition 4.3 A *query* is a pair $Q = (\varphi, \vec{x})$ where φ is an $\mathcal{L}_\mathcal{R}$-formula and $\vec{x}$ is a fixed enumeration of $\mathit{fr}(\varphi)$. The set $\mathit{fr}(\varphi)$ is also called the set of free variables of Q and denoted in the form $\mathit{fr}(Q)$.

Definition 4.4 Let $\mathcal{D}$ be a relational document structure, let $Q = (\varphi, \vec{x})$ be a query where $\vec{x} = \langle x_1, \ldots, x_n \rangle$. A sequence $\vec{d} = \langle d_1, \ldots, d_n \rangle$ of nodes of $\mathcal{D}$ is an *answer* to Q in $\mathcal{D}$ iff $\mathcal{D} \models \varphi[d_1, \ldots, d_n]$. The set

$$A_\mathcal{D}(Q) := \{\langle d_1, \ldots, d_n \rangle \in D^n \mid \mathcal{D} \models \varphi[d_1, \ldots, d_n]\}$$

is called the *complete set of answers* for Q in $\mathcal{D}$.

Equivalently, an answer $\langle d_1, \ldots, d_n \rangle$ can be considered as a partial variable assignment mapping x_i to d_i for $i = 1, \ldots, n$. Both perspectives will be used in the sequel.

The efficient computation of the complete set of answers to a given query can be considered as the principal problem of an IR system. In this paper we concentrate on a particular subclass of queries.

4.2 Tree queries

It is natural to assume that most queries aim to retrieve subtrees of a particular form from the database. As long as matters of universal quantification are ignored, the tree queries introduced in the following definition are a canonical choice for this retrieval task.

Definition 4.5 A *tree query* is a query $Q = (\psi \wedge c, \vec{x})$ where ψ is a conjunction of atomic $\mathcal{L}$-formulae and c is a conjunction of atomic constraints with $\mathit{fr}(c) \subseteq \mathit{fr}(\psi)$ such that the following condition is satisfied: there exists a variable $x \in \mathit{fr}(Q)$, called the *root* of Q, such that for each $y \in \mathit{fr}(Q)$ there exists a unique sequence of variables $x = x_0, \ldots, x_n = y$ $(n \geq 0)$ where ψ contains subformulae $x_i \lhd^{(+)} x_{i+1}$ for $0 \leq i \leq n-1$. A tree query $Q = (\psi \wedge c, \vec{x})$ is *rigid* if ψ does not contain any formula of the form $y \lhd^{+} z$. A tree query Q is *labeling-complete* if for each $x \in \mathit{fr}(Q)$ there exists either a formula of the form w *in* x in Q or a formula of the form $M(x)$ $(M \in \Gamma)$.

The following lemma follows immediately from Definition 4.5.

Lemma 4.6 *Let $Q = (\psi \wedge c, \vec{x})$ be a tree query. Then*

1. *ψ does not have any $\lhd^{(+)}$-cycle, i.e., there is no sequence of variables $x_0, \ldots, x_n$ $(n \geq 0)$ such that ψ contains subformulae $x_i \lhd^{(+)} x_{i+1}$ for $0 \leq i \leq n-1$ and a subformula $x_n \lhd^{(+)} x_0$,*
2. *the root of Q is unique, and*
3. *for each $y \in \mathit{fr}(Q)$ there exists at most one formula in ψ of the form $x \lhd^{(+)} y$.*

Definition 4.7 A tree query $Q = (\psi \wedge c, \vec{x})$ is *inconsistent* if

1. φ contains a formula w *in* x, and a formula of the form $M(x)$ or $x \lhd^{(+)} y$ at the same time, or
2. if φ contains two formulae $M(x)$ and $M'(x)$ for $M' \neq M \in \Gamma$.

Clearly, inconsistent tree queries are unsatisfiable. Below we shall see that every consistent tree query is satisfiable if we do not restrict the interpretation of relation symbols in $\mathcal{R}$. Henceforth, with a tree query we always mean a consistent tree query. Since tree queries have a tree shape, standard notions from trees can be used for describing their structure:

Definition 4.8 Let $Q = (\psi \wedge c, \vec{x})$ be a tree query. A variable $y \in \mathit{fr}(Q)$ is called a rigid (resp. soft) *child* of $x \in \mathit{fr}(Q)$ in Q iff ψ contains a formula $x \lhd y$ (resp. $x \lhd^+ y$). In this situation, x is called a rigid (soft) *parent* of y in Q. Two distinct children y_1, y_2 of a variable x in Q are said to be *siblings* in Q. A variable $y \in \mathit{fr}(Q)$ is a *reflexive descendant* of x in Q iff there exists a chain $x = x_0, \ldots, x_k = y$ of length $k \geq 0$ such that x_{i+1} is a child (of either type) of x_i in Q, for $i = 0, \ldots, k-1$. A *partial path* of Q is a chain $x_0, \ldots, x_k$ of length $k \geq 0$ starting at the root of Q such that x_{i+1} is a child (of either type) of x_i in Q for $i = 0, \ldots, k-1$. A *path* of Q is a maximal partial path. Let $x \in \mathit{fr}(Q)$. The *height* $h_Q(x)$ of x with respect to Q is defined as follows: $h_Q(x) := 0$ iff x does not have any child in Q, and $h_Q(x) := \max\{h_Q(y) + 1 \mid y \text{ is a child of } x \text{ in } Q\}$ otherwise.

Note that x is always a reflexive descendant of x in Q, for each $x \in \mathit{fr}(Q)$. In the following sections three subclasses of tree queries (simple, local and partially ordered tree queries) will be considered, each obtained by restricting the classes of constraints that may be used in queries.

Simple and local tree queries

Definition 4.9 Let $Q = (\psi \wedge c, \vec{x})$ be a tree query. A constraint $r(x_0, \ldots, x_n)$ of c is *Q-simple* iff either r is unary, or r is binary and x_1 (or x_0) is a child of x_0 (resp. x_1) in Q. A constraint $r(x_0, \ldots, x_n)$ of c is *Q-local* iff Q has a variable x with children $y_1, \ldots, y_h$ such that $\{x_0, \ldots, x_n\} \subseteq \{x, y_1, \ldots, y_h\}$. We say that Q is a *simple (local) tree query iff each constraint of c is Q-simple (Q-local).*

It is important to note that "Q-simplicity" is a purely syntactic concept that just restricts the pairs of variables of Q that can be related in constraints. From a semantic point of view, Q-simple constraints can talk about arbitrary unary or binary relations over $\mathcal{D}$. Some examples are

1. the relation "*top-most M-descendant*", $\mathit{tm}_M^{\mathcal{D}}(d, e)$, which expresses that node e is a descendant of d with label M such that there is no node between d and e with label M in $\mathcal{D}$,

2. vertical distance relations such as, e.g, "$\mathit{max\text{-}distance}(k)(d, e)$" which expresses that there are at most $k-1$ nodes between d and e,

3. unary relations encoding typing information, e.g. taxonomic information, such as being a document of a specific type, or a node describing a year,

4. unary relations expressing that a certain attribute is defined for the node and has a particular value (e.g., "*gender* = *female*" if documents are parse trees for natural language expressions).

With Q-local constraints we can even express arbitrary relations in principle. However, we can only constrain the children (plus parent) of a common parent variable x of Q. Some examples are

5. comparisons (e.g., "node d_i dominates a larger part of the database than node d_j"),
6. conditions that express that the text parts dominated by d_i and d_j stand in a certain syntactic relationship, such as similarity or containment,
7. conditions that express that the text parts dominated by d_i and d_j stand in a certain semantic relationship (e.g. chronological comparison of dates, comparison of amounts of money etc.).

An important class of Q-local constraints are the constraints $y_i <_{lr} y_j$ expressing that two nodes stand in the left-to-right ordering relation $<_{lr}^{D}$ of $\mathcal{D}$. A formula $y_i <_{lr} y_j$ will be called a *left-to-right ordering constraint.*

Partially ordered tree queries

Definition 4.10 A tree query $Q = (\psi \wedge c, \vec{x})$ is *partially ordered* if each constraint of c is either Q-simple or a left-to-right ordering constraint, and if the subset c_{lr} of left-to-right ordering constraints in c satisfies the following properties:

1. for each constraint $y_i <_{lr} y_j$ in c_{lr} the variables y_i and y_j are siblings with respect to Q,
2. c_{lr} does not have a cycle of the form $y_0 <_{lr} \cdots <_{lr} y_n <_{lr} y_0$.

The tree query $Q = (\psi \wedge c, \vec{x})$ is *linearly ordered* if the set of left-to-right constraints specifies a linear ordering for the set of children of each variable x of Q.

Note that in particular each partially ordered tree query is a local tree query. The following lemma shows—in a sense to be made precise—that for local tree queries the possible instantiations of the reflexive descendants of a variable $x \in \mathit{fr}(Q)$ in answers only depend on the instantiation of x, but not on the instantiation of the ancestors of x in Q. The lemma will play an essential role for the techniques suggested in the following sections.

Lemma 4.11 *Let $Q = (\psi \wedge c, \vec{x})$ be a local tree query. Let y be a child of x in Q, let Y denote the set of proper descendants of y in Q, and let $Z = \mathit{fr}(Q) \setminus (Y \cup \{y\})$. Let $\langle y_1, \ldots, y_r \rangle$ and $\langle z_1, \ldots, z_s \rangle$ denote enumerations of Y and Z respectively. Assume that $\vec{x}$ has the form $\langle z_1, \ldots, z_s, y, y_1, \ldots, y_r \rangle$. If Q has two answers*

$$\langle d_1, \ldots, d_s, \quad d \quad, e_1, \ldots, e_r \rangle$$
$$\langle d'_1, \ldots, d'_s, \quad d \quad, e'_1, \ldots, e'_r \rangle$$

that coincide on y, then

$$\langle d_1, \ldots, d_s, \quad d \quad, e'_1, \ldots, e'_r \rangle$$
$$\langle d'_1, \ldots, d'_s, \quad d \quad, e_1, \ldots, e_r \rangle$$

are answers to Q as well.

Proof. This follows immediately from the fact that $\psi \wedge c$ does not have any atomic subformula that contains variables from Y and Z at the same time. □

4.3 Answers as pseudo-homomorphisms

Tree queries can be represented in the form of "generalized" tree structures. This description might offer a basis for a graphical user interface for queries and leads to a second, algebraic picture of the query evaluation process where answers essentially behave like homomorphisms, like elaborated in the Tree Matching formalism [Kil92] and in [Meu98].

Definition 4.12 Let $Q = (\psi \wedge c, \vec{x})$ be a tree query. The *query tree* for Q is the relational structure $\mathcal{D}_Q = (X_S, X_T, X_U, \rightharpoonup, \overset{+}{\rightharpoonup}, \mathit{Lab}, I)$ with the following components:

- X_S is the set of all variables x occurring in ψ such that ψ has a subformula of the form $M(x)$ or $x \lhd^{(+)} y$,
- X_T is the set of all variables x occurring in ψ such that ψ has a subformula w *in* x,
- $X_U := \mathit{fr}(Q) \setminus (X_T \cup X_S)$,
- for $x, y \in X := X_S \cup X_T \cup X_U$ we have $x \rightharpoonup y$ iff $x \lhd y \in \psi$,

– for $x, y \in X$ we have $x \overset{+}{\rightharpoonup} y$ iff $x \lhd^{+} y \in \psi$,

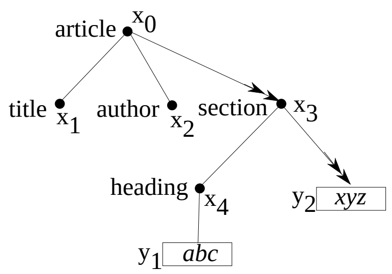
– for $x \in X_S$ we have $Lab(x) = M$ iff $M(x) \in \psi$,

– for $x \in X_T$ we have $Lab(x) = \{w_1, \ldots, w_k\}$ where $\{w_1, \ldots, w_k\} = \{w \in \Sigma^* \mid w \textit{ in } x \in \psi\}$,

– I is the interpretation function where $\langle y_1, \ldots, y_k\rangle \in I(r)$ iff $r(y_1, \ldots, y_k) \in c$.

Note that consistency guarantees that $X_S \cap X_T = \emptyset$, and that Lab is well-defined for nodes in X_S. By part 1 of Lemma 4.6, the transitive closure of $\overset{+}{\rightharpoonup} \cup \rightharpoonup$ is cycle-free. Intuitively, $\mathcal{D}_Q = (X_S, X_T, X_U, \rightharpoonup, \overset{+}{\rightharpoonup}, Lab, \vec{x})$ can be considered as "generalized" relational document structure with edges of two types. Edges of the form $x \rightharpoonup y$ are called *rigid*, edges of the form $x \overset{+}{\rightharpoonup} y$ are called *soft*.[2] In contrast to document structures, a query tree $\mathcal{D}_Q$ may have unlabeled nodes if Q is not labeling-complete.

Example 4.13 The query tree



is a query tree for the tree query with the atomic formulae $article(x_0)$, $x_0 \lhd x_1$, $title(x_1)$, $x_0 \lhd x_2$, $author(x_2)$, $x_0 \lhd^{+} x_3$, $section(x_3)$, $x_3 \lhd x_4$, $heading(x_4)$, $x_4 \lhd y_1$, $abc \textit{ in } y_1$, $x_3 \lhd^{+} y_2$, $xyz \textit{ in } y_2$, and the sequence of variables $\langle x_0, x_1, x_2, x_3, x_4, y_1, y_2\rangle$. This query may be used to retrieve authors and titles of articles that contain a section with the heading "*abc*", with the word "*xyz* " occurring in the section.

Lemma 4.14 *Each tree query where relation symbols do not have a fixed interpretation is satisfiable.*

Proof. Let $Q = (\psi \wedge c, \vec{x})$ be a tree query. We may modify the query tree for Q as follows. Each unlabeled node receives a fixed label $M \in \Gamma$. Every labeled textual node x receives as label the concatenation of the words in its label: $Lab(X) = w_1 \circ \ldots \circ w_k$, where $\{w_1, \ldots, w_k\} = \{w \in \Sigma^* \mid w \textit{ in } x \in \psi\}$. Each soft edge is

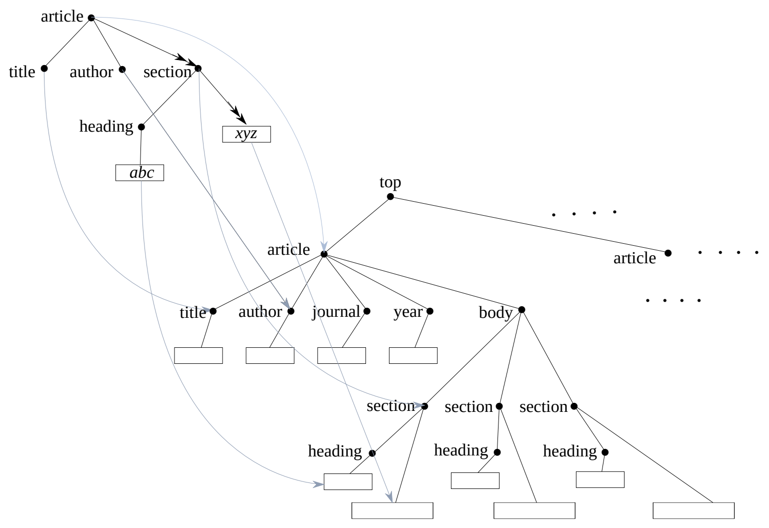
[2]Edges of the form $x \overset{+}{\rightharpoonup} y$ in a query tree should not be confused with the transitive closure $\rightharpoonup_D^{+}$ of edges $\rightharpoonup_D$ in a relational document structure $\mathcal{D}$.

treated as a rigid edge. Obviously in this way a relational document structure $\mathcal{D}$ is obtained such that $\psi \wedge c$ holds in $\mathcal{D}$ under each variable assignment that maps each element of $\vec{x}$ to itself. $\square$

Definition 4.15 Let $\mathcal{D}_Q = (X_S, X_T, X_U, \rightharpoonup_Q, \overset{+}{\rightharpoonup}_Q, Lab_Q, I_Q)$ be the query tree of the tree query Q, let $\mathcal{D} = (D_S, D_T, \rightharpoonup_D, Lab_D, I_D)$ be a relational document structure. Let $X_Q := X_S \cup X_T \cup X_U$. A *pseudo-homomorphism* from $\mathcal{D}_Q$ in $\mathcal{D}$ is a mapping $\nu : X_Q \to D$ such that the following conditions are satisfied:

- for all $x \in X_S$ always $\nu(x) \in D_S$,
- for all $x \in X_T$ always $\nu(x) \in D_T$,
- for all $x, y \in X_Q$: $x \rightharpoonup_Q y$ implies $\nu(x) \rightharpoonup_D \nu(y)$,
- for all $x, y \in V^Q$: $x \overset{+}{\rightharpoonup}_Q y$ implies $\nu(x) \rightharpoonup_D^+ \nu(y)$,
- for all $x \in X_S$: $Lab_Q(x) = M$ implies $Lab_D(\nu(x)) = M$,
- for all $x \in X_T$: $w \in Lab_Q(x)$ implies $Lab_D(\nu(x))$ contains w as substring,
- for all $r \in \mathcal{R}$, for each sequence $(y_1, \ldots, y_k)$ of variables of Q: $\langle y_1, \ldots, y_k \rangle \in I_Q(r)$ implies $\langle \nu(y_1), \ldots, \nu(y_k) \rangle \in I_D(r)$.

Example 4.16 The following figure shows a pseudo-homomorphism for the tree query introduced in Example 4.13 and the relational document structure depicted in Example 3.4.



The following lemma shows that pseudo-homomorphisms and answers to a given tree query are equivalent notions. The proof can be found in the Appendix.

Lemma 4.17 *Let* $\mathcal{D}_Q = (X_S, X_T, X_U, \rightharpoonup_Q, \overset{+}{\rightharpoonup}_Q, Lab_Q, I_Q)$ *be the query tree of the tree query* Q*, let* $\mathcal{D} = (D_S, D_T, \rightharpoonup_D, Lab_D, I_D)$ *be a relational document structure. A mapping* $\nu : \mathrm{fr}(Q) \rightarrow D$ *is a pseudo-homomorphism from* $\mathcal{D}_Q$ *in* $\mathcal{D}$ *iff* ν *is an answer to* Q *in* $\mathcal{D}$*.*

4.4 Partially ordered tree queries and Tree Matching

We mentioned in the introduction that the present formalism can be understood as a variant and generalization of Kilpeläinen's Tree Matching formalism [Kil92]. In this subsection we briefly comment on this point.

In Kilpeläinen's formalism, both patterns (i.e., queries) and targets are finite, ordered labeled trees. Kilpeläinen studies ten distinct variants for formalizing the notion of "inclusion" (homomorphic embedding) between pattern and target. A basic difference is that between unordered and ordered tree inclusion problems. In the (un)ordered case the left-to-right ordering of nodes has (not) to be respected under an embedding. In our formalism this corresponds to the difference between unordered tree queries on the one side and linearly ordered tree queries on the other side.

Both for the ordered and the unordered case, five specific classes are considered. For so-called *tree inclusion*, the homomorphic embedding has only to respect labels and ancestorship. For *path inclusion*, parent relationship has to be respected as well. Basically, path inclusion problems thus correspond to rigid queries, tree inclusion problems correspond to tree queries with soft edges only. For *region inclusion*, which restricts path inclusion, any child of a target node that has a left sibling and a right sibling that both belong to the image of the embedding function has to belong to this image as well. We do not have a similar construct in our formalism. For *child inclusion*, which restricts region inclusion, the number of children of inner nodes has to be respected. Again we do not have a similar construct.

When we ignore region inclusion and child inclusion, which seem not important for database applications, our formalism is more flexible than Tree Matching since in a partially ordered tree query we may specify an arbitrary partial ordering between the children of a query node, and we can also have rigid edges and soft edges at the same time. In this sense, the present formalism generalizes Tree Matching. However, there are also two subtle differences:

- Kilpeläinen's homomorphic embeddings are always assumed to be injective, we do not impose such a restriction in our formalism.
- A relation is "preserved" in Kilpeläinen's sense under a mapping h if it is preserved in both directions. For example, a mapping is said to preserve ancestorship if, for all nodes x, y of the pattern, x is an ancestor of y *if and only if* $h(x)$ is an ancestor of $h(y)$. We only demand the implication from left to right, i.e., the "only if" direction.

These innocent differences are responsible for the phenomenon that Kilpeläinen's unordered tree inclusion problem is NP-complete even in the decision version (cf. [Kil92]) whereas all the complexity results obtained here are polynomial. Since unordered tree inclusion problems represent the most natural variant of Tree Matching, the avoidance of the above intractability result can be considered as a major advantage of the present formalism.

5 Representing complete sets of answers for tree queries

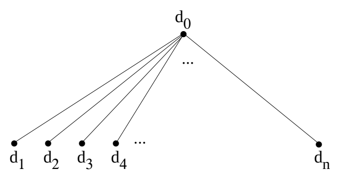
This section is devoted to the problem of finding a suitable presentation for the complete set of answers to a given tree query. As we demonstrate below, the number of answers to a tree query Q may be exponential in the size of Q. Hence an explicit enumeration leads to exponential-time behavior in the worst case. Quite generally a naive enumeration will also suffer from many redundancies since different answers may have several common sub-nodes. This makes it difficult to extract useful information from the sequence of answers.

The question arises if there is a more compact and organized way of representing all answers, with reasonable space and time requirements. In the first subsection we introduce the concept of a "complete answer formula" for a simple tree query. The "complete answer aggregates" that are introduced in the second subsection yield a physical representation of complete answer formulae. A complete answer aggregate yields a full representation of all answers that is quadratic in the size of the database for simple tree queries. For rigid queries, as well as for labeling-complete queries over non-recursive databases, the size is linear. In subsections three and four these notions and results are extended to local tree queries and to ordered tree queries. Computational aspects are postponed to Section 6.

5.1 Complete answer formulae for simple tree queries

In order to introduce our representation technique we fix a simple tree query, Q, and a relational document structure $\mathcal{D}$ with set of nodes D. If Q has q variables, in the worst case the total number of answers to Q is of order $\mathcal{O}(|D|^q)$, even for rigid queries.

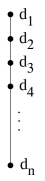
Example 5.1 Let $\mathcal{D}$ have the following form (we ignore labels and textual contents).



The rigid query Q of the form $(x \lhd y_1 \wedge \ldots \wedge x \lhd y_q, \langle x, y_1, \ldots, y_q \rangle)$ has n^q answers in $\mathcal{D}$.

The example shows that nodes with a high branching degree may lead to an explosion of the number of answers. For queries with soft edges an orthogonal potential source of problems is deep nesting:

Example 5.2 Let $\mathcal{D}$ have the following form.



The query Q of the form $(x_1 \lhd^+ x_2 \wedge \ldots \wedge x_{q-1} \lhd^+ x_q, \langle x_1, \ldots, x_q \rangle)$ has $\binom{n}{q}$ answers in $\mathcal{D}$.

How can we avoid an enumeration of all answers? As a starting point, we take a logical perspective. The complete set of answers to Q in $\mathcal{D}$ can be represented as a formula in disjunctive normal form

$$\bigvee_{(d_1,\ldots,d_q) \in A_{\mathcal{D}}(Q)} (\bigwedge_{i=1}^{q} x_i = d_i),$$

It is well-known that the size of the disjunctive normal form of a formula of propositional logic may be exponential in the size of the original formula. Even if we do not have anything like an "original" formula here, an obvious idea is to look for formulae that are logically equivalent to the disjunction of all answers but of smaller size, and to use a shared representation for multiple occurrences of the same subformula. Of course, from a practical point of view the formulae must offer a transparent view of all answers, and given the formula it should be possible to generate each particular answer without computational effort. Before we introduce a suitable class of formulae, let us illustrate the basic idea using the above examples.

Example 5.3 The set of all answer in Example 5.1 can be encoded as a formula of size $\mathcal{O}(q \cdot n)$ of the form

$$x = d_0 \wedge \bigwedge_{i=1}^{q} (\bigvee_{j=1}^{n} y_i = d_j).$$

The set of all answers in Example 5.2 can be encoded as a formula of the form

$$\bigvee_{i_1=1}^{n-q+1} (x_1 = d_{i_1} \wedge \bigvee_{i_2=d_{i_1}+1}^{n-q+2} (x_2 = d_{i_2} \wedge \bigvee_{i_3=d_{i_2}+1}^{n-q+3} (\ldots \bigvee_{i_q=d_{i_{q-1}}+1}^{n} (x_q = d_{i_q}) \ldots)))$$

Given this formula, each answer to Q can be immediately obtained in the following way. Select a possible value d_{i_1} for x_1 in the outermost disjunction. Each possible choice leads to a specific subdisjunction that gives possible values for x_2. Continuing in the same way, the choice of a value d_{i_k} for x_k $(k < q)$ always determines a new subdisjunction that determines a set of possible values for x_{k+1}. In more detail, a value $x_k = d_{i_k}$ always implies that the value for x_{k+1} can recruit from $\{d_{i_k+1}, \ldots, d_{n-q+k+1}\}$.

Let ϵ stand for the empty sequence (). The following definition generalizes the above type of representation, introducing a class of formulae that may be used for "dependent instantiation" of variables, given Q. The idea is to instantiate the variables of Q in a top-down manner, where the sets of possible values of descendant variables with respect to Q depend on the chosen instantiations of the ancestor variables.

Definition 5.4 Let $Q = (\psi \wedge c, \vec{x})$ be a simple tree query, let $x \in \mathit{fr}(Q)$. The set of *dependent Q-instantiation formulae for x* is inductively defined as follows. First assume that $h_Q(x) = 0$. For each non-empty set $D_x \subseteq D$, the formula

$$\Delta_{\epsilon}^{(x)} :\equiv \bigvee_{d \in D_x} x = d$$

is a dependent Q-instantiation formula for x. Now assume that $h_Q(x) > 0$. Let $\emptyset \neq D_x \subseteq D$. For each $d \in D_x$ and each child y of x in Q, let $\Delta_{(d)}^{(x,y)}$ be a dependent

Q-instantiation formula for y. Then

$$\Delta_\epsilon^{(x)} :\equiv \bigvee_{d \in D_x} (x = d \wedge \bigwedge_{y \textit{ child of } x \textit{ in } Q} \Delta_d^{(x,y)})$$

is a dependent Q-instantiation formula for x. There are no other dependent Q-instantiation formulae for x besides those defined above. The set D_x is called the set of *target candidates for* x in $\Delta_\epsilon^{(x)}$.

Note that the notion of a dependent Q-instantiation formula is defined in a purely syntactical way and does not refer to an answer. In the sequel we use expressions of the form $\Delta_{(d_0,\ldots,d_{k-1})}^{(x_0,\ldots,x_k)}$ for refering to subformulae of a dependent Q-instantiation formula $\Delta_\epsilon^{(x_0)}$. These subformulae are inductively defined as follows: Assume that $\Delta_{(d_0,\ldots,d_{k-1})}^{(x_0,\ldots,x_k)}$ is a dependent Q-instantiation subformula of $\Delta_\epsilon^{(x_0)}$ of the form

$$\bigvee_{d \in D_{x_k}} (x_k = d \wedge \bigwedge_{y \textit{ child of } x_k \textit{ in } Q} \Delta_{(d)}^{(x_k,y)})$$

where $k \geq 0$. If x_{k+1} is a child of x_k in Q and $d_k \in D_{x_k}$, then $\Delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_{k+1})}$ denotes $\Delta_{(d_k)}^{(x_k,x_{k+1})}$. Furthermore each disjunct

$$x_k = d \wedge \bigwedge_{y \textit{ child of } x_k \textit{ in } Q} \Delta_{(d)}^{(x_k,y)}$$

is written in the form $\delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$. As an immediate consequence of these definitions we obtain

Remark 5.5 Modulo associativity and commutativity of "$\wedge$" and "$\vee$", each formula $\delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$ is uniquely determined by its subformulae of the form $\Delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k,y)}$ where y is a child of x_k in Q. Similarly each subformula of the form $\Delta_{(d_0,\ldots,d_{k-1})}^{(x_0,\ldots,x_{k-1},x_k)}$ is uniquely determined by its subformulae of the form $\delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$.

The following definition captures the notion of incrementally instantiating a dependent Q-instantiation formula in a top-down manner.

Definition 5.6 The set of *partial* (resp. *total*) *instantiations* of a dependent Q-instantiation formula $\Delta_\epsilon^{(x_0)}$ is inductively defined as follows: the empty set "$\emptyset$" is a partial instantiation of $\Delta_\epsilon^{(x_0)}$. Assume that ν is a partial instantiation of $\Delta_\epsilon^{(x_0)}$. If $\Delta_\epsilon^{(x_0)}$ has a subformula of the form $\delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$, if $\{\langle x_i, d_i \rangle \mid 1 \leq i \leq k-1\} \subseteq \nu$ and ν does not have a pair of the form $\langle x_k, d \rangle$ $(d \in D)$, then $\nu \cup \{\langle x_k, d_k \rangle\}$ is a partial instantiation of $\Delta_\epsilon^{(x_0)}$. There are no other partial instantiations besides those defined by the above rules. A partial instantiation ν of $\Delta_\epsilon^{(x_0)}$ is a *total instantiation* iff ν contains a pair $\langle x, d \rangle$ for every $x \in \mathit{fr}(\Delta_\epsilon^{(x_0)})$.

Lemma 5.7 *Let Q be a simple tree query, let $\Delta_\epsilon^{(x_0)}$ be a dependent Q-instantiation formula. Then every partial instantiation of $\Delta_\epsilon^{(x_0)}$ can be extended to a total instantiation of $\Delta_\epsilon^{(x_0)}$. The set of total instantiations of $\Delta_\epsilon^{(x_0)}$ is non-empty.*

The lemma can be proven by a trivial induction on $h_Q(x_0)$.

Lemma 5.8 *Let Q be a simple tree query, let $\Delta_\epsilon^{(x_0)}$ be a dependent Q-instantiation formula for x_0, let $d_0, \ldots, d_k$ be elements of D. Then the following conditions are equivalent:*

1. *$\Delta_\epsilon^{(x_0)}$ has a subformula $\Delta_{(d_0,\ldots,d_{k-1})}^{(x_0,\ldots,x_k)}$ where d_k is a target candidate for x_k,*
2. *$\Delta_\epsilon^{(x_0)}$ has a subformula of the form $\delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$,*
3. *Q has formulae $x_0 \lhd^{(+)} x_1, \ldots, x_{k-1} \lhd^{(+)} x_k$ and $\{\langle x_i, d_i\rangle \mid 0 \leq i \leq k\}$ is a partial instantiation of $\Delta_\epsilon^{(x_0)}$.*

The proof of Lemma 5.8 is simple and can be found in the Appendix. We now come to the central definition of this section.

Definition 5.9 Let Q be a simple tree query with root x_0. A dependent Q-instantiation formula $\Delta_\epsilon^{(x_0)}$ for x_0 is called a *complete answer formula* for Q iff each answer to Q is a total instantiation of $\Delta_\epsilon^{(x_0)}$ and vice versa.

Complete answer formulae will be denoted in the form Δ_Q. Some of the following formulations become simpler when introducing the falsum "$\perp$" as an additional dependent Q-instantiation formula. By convention, "$\perp$" does not have any instantiation. We may now give the first kernel result of this paper.

Theorem 5.10 *For each simple tree query $Q = (\psi \wedge c, \vec{x})$ and each relational document structure $\mathcal{D}$ there exists a complete answer formula Δ_Q which is unique modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Proof. First assume that Q does not have any answer in $\mathcal{D}$. Then "$\perp$" is a complete answer formula for Q. It follows from Lemma 5.7 that Q does not have another complete answer formula. Assume now that Q has at least one answer. Since we later see how to compute a complete answer formula Δ_Q for Q (cf. Sections 6.3 and 6.5) we only prove the uniqueness part here. Let x_0 be the root of Q, let $\Delta_\epsilon^{(x_0)}$ and $\Lambda_\epsilon^{(x_0)}$ be complete answer formulae for Q. Lemma 5.7 and Lemma 5.8

show that $\Delta_\epsilon^{(x_0)}$ has a subformula $\delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$ iff $\Lambda_\epsilon^{(x_0)}$ has a subformula $\lambda_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$. Starting at the subformulae with maximal k it is then trivial to prove by "inverse" induction using Remark 5.5 that corresponding formulae $\delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$ and $\lambda_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$ are equal modulo associativity and commutativity of "$\wedge$" and "$\vee$". It follows that $\Delta_\epsilon^{(x_0)}$ and $\Lambda_\epsilon^{(x_0)}$ are equal modulo associativity and commutativity of "$\wedge$" and "$\vee$". $\square$

Since we want to obtain a representation where multiple occurrences of the same subformula are shared, the following simple observation is crucial. The proof, which strongly depends on Lemma 4.11, can be found in the Appendix.

Lemma 5.11 *Let Δ_Q be a complete answer formula for the simple tree query Q. Then two subformulae of Δ_Q of the form $\delta_{(d_0,\ldots,d_{k-1},d_k)}^{(x_0,\ldots,x_{k-1},x_k)}$ and $\delta_{(d'_0,\ldots,d'_{k-1},d_k)}^{(x_0,\ldots,x_{k-1},x_k)}$ are always identical modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

5.2 Complete answer aggregates for simple tree queries

Our next aim is to give a compact physical representation of complete answer formula. Lemma 5.11 shows that for each pair (x_k, d_k) (where $x_k \in fr(Q)$ and $d_k \in D$) all subformulae of Δ_Q of the form $\delta_{(d_0,\ldots,d_{k-1},d_k)}^{(x_0,\ldots,x_{k-1},x_k)}$ are identical. We shall write them in the form $\delta_{x_k}(d_k)$. In the physical representation, all occurrences of a subformula $\delta_x(d)$ are shared and represented as a field $Agg_x[d]$ of a record[3] Agg_x assigned to the variable x.

Definition 5.12 Let $Q = (\psi \wedge c, \vec{x})$ be a simple tree query. An *aggregate* for Q is a family Agg_Q of records, $\{Agg_x \mid x \in \vec{x}\}$. Each record is composed of a finite number of fields with indices $d \in D$, denoted $Agg_x[d]$. For each child y of x in Q, the field $Agg_x[d]$ contains a list of pointers, $Agg_x[d, y]$. Each pointer in a list $Agg_x[d, y]$ points to a field $Agg_y[e]$ of the record Agg_y. Distinct pointers of $Agg_x[d, y]$ point to distinct fields.

In Examples 5.15 and 5.16 graphical representations for aggregates may be found. Since we are concerned in this section with the size of answers we will define the

[3]In some contexts, the data structure that is used here, with an open number of fields that are accessed by arbitrary keys, are called "dictionaries" and distinguished from "records" (which have a fixed number of fields). Since the terminus "dictionary" is preoccupied to a certain extend in our context we prefer to ignore this difference here.

size of an aggregate as the number of pointers of the aggregate. Modulo a constant factor this value reflects the storage space needed for an aggregate.

Definition 5.13 Let $Q = (\psi \wedge c, \vec{x})$ be a simple tree query, let Δ_Q denote the complete answer formula for Q. A *complete answer aggregate* for Q is a Q-aggregate $Agg_Q = \{Agg_x \mid x \in \vec{x}\}$ that satisfies the following conditions:

1. a record Agg_x has a subfield $Agg_x[d]$ iff Δ_Q has a subformula $\delta_x(d)$,
2. a list $Agg_x[d, y]$ has a pointer to a field $Agg_y[e]$ iff $\delta_x(d)$ contains a subformula of the form $\delta_y(e)$.

Remark 5.14 Ignoring the trivial case of an unsatisfiable query it is easy to see that a complete answer formula Δ_Q for a simple tree query Q uniquely determines the corresponding complete answer aggregate Agg_Q. Conversely, given a complete answer aggregate Agg_Q we may reconstruct the complete answer formula Δ_Q in the following way: to obtain Δ_Q,

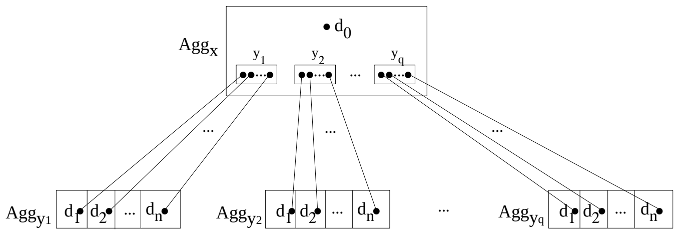
– read the record Agg_x of the root x of Q as the disjunction of the formulae associated with the subfields $Agg_x[d]$,

– associate with each field $Agg_x[d]$ the conjunction of $x = d$ with the formulae associated with the pointer lists $Agg_x[d, y]$,

– associate with each list of pointers $Agg_x[d, y]$ of a field $Agg_x[d]$ the disjunction of the formulae associated with the address fields of the pointers.

The correspondence between the two concepts should become more obvious with the following examples.

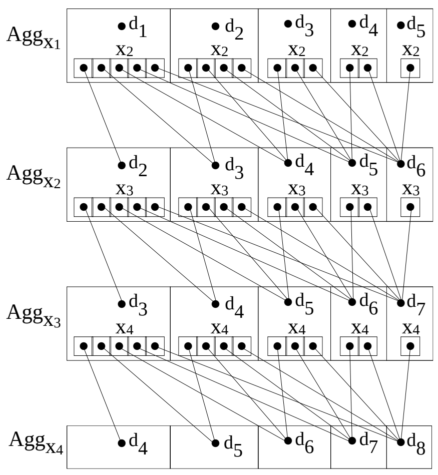
Example 5.15 The complete answer aggregate for the first formula in Example 5.3 (an encoding of all answers to the rigid tree query given in Example 5.1) can be depicted as follows.



The root variable x can only be instantiated with d_0. All other variables can be instantiated with each of the nodes $d_1, \ldots, d_n$.

In this example, the number of pointers of the aggregate is $q \cdot n$. Hence the size of the aggregate is of order $\mathcal{O}(q \cdot n)$.

Example 5.16 For the special case $q = 4$ and $n = 8$ the complete answer aggregate for the second formula in Example 5.3 (an encoding of all answers to the tree query in Example 5.2) has the following form.



For arbitrary $n \geq q$, each record Agg_{x_q} contains $n - q + 1$ target candidates. The first target candidate in each record Agg_{x_i} (apart from the last field Agg_{x_q}, which contains no pointers at all) has $n-q+1$ pointers to target candidates in $Agg_{x_{i+1}}$, the last target candidate contains only one pointer. Hence each record Agg_{x_i} contains $1 + \ldots + (n - q + 1) = \frac{(n-q+1)(n-q+2)}{2}$ pointers, apart from the target candidates in the leaf record Agg_{x_q}. Therefore the total number of pointers in the aggregate is $(q - 1) \cdot \frac{(n-q+1)(n-q+2)}{2}$, hence of order $\mathcal{O}(q \cdot (n - q)^2)$.

We show now that the size of a complete answer aggregate for a rigid simple tree query, Q, is linear both in the size of the query and the database. In the sequel, let $|Q|$ denote the number of symbols of Q. This means in particular that the number of variables of Q and the number of atomic constraints of Q is bounded by $|Q|$. With $|D|$ we denote the cardinality of D.

Theorem 5.17 *Let $\mathcal{D}$ be a relational document structure and let Q be a rigid simple tree query. Then the size of the complete answer aggregate for Q is of order $\mathcal{O}(|Q| \cdot |D|)$.*

Proof. The complete answer aggregate Agg_Q for Q contains $\leq |Q|$ records Agg_x, the total number of fields $Agg_x[d]$ is bounded by $|Q| \cdot |D|$. For a fixed field $Agg_y[e]$ there is at most one pointer ending at $Agg_y[e]$. In fact, each such pointer starts at a field of the form $Agg_x[d]$ where x is the parent of y in Q: the definition of the complete answer aggregate implies that the complete answer formula, Δ_Q, has a formula $\delta_x(d)$ with subformula $\delta_y(e)$. Lemma 5.8 and Lemma 5.7 show that Δ_Q has a total instantiation ν mapping x to d and y to e. Since ν is an answer to Q and y is a rigid child of x in Q it follows that d is the unique parent of e in $\mathcal{D}$. We have seen that the total number of pointers is bounded by $|Q| \cdot |D|$. It follows that the total size of Agg_Q is of order $\mathcal{O}(|Q| \cdot |D|)$. $\square$

Theorem 5.18 *Let $\mathcal{D}$ be a non-recursive relational document structure and let Q be a labeling-complete tree query. Then the size of the complete answer aggregate for Q is of order $\mathcal{O}(|Q| \cdot |D|)$.*

Proof. Similar to the previous proof. Again, for a fixed field $Agg_y[e]$ there is at most one pointer ending at $Agg_y[e]$. In fact, if x denotes the parent of y, then the query contains a formula $M(x)$ and there is at most one ancestor d of e in $\mathcal{D}$ with label M. $\square$

Theorems 5.17 and 5.18 depend on the fact that in the situation of these theorems for each field $Agg_x[d]$ of the complete answer aggregate there exists at most one vertical pointer that points to $Agg_x[d]$. If we conceive the aggregate as a graph, with the target candidates as nodes and the pointers as edges, then the resulting graph is a forest. Some nodes of the relational document structure may appear more than once in this forest, due to the fact that they appear as target candidates in more than one record. If we use an equivalence relation to collapse these duplicates, the resulting forest is a subforest of the relational document structure.
For arbitrary tree queries and databases, a given field can serve as the address of more than one pointer (see Example 5.16 for an illustration). Hence the graph induced by the aggregate is not necessarily a tree and the maximal size also depends on the maximal length of a document path, denoted $h_{\mathcal{D}}$:

Theorem 5.19 *Let $\mathcal{D}$ be a relational document structure and let Q be a simple tree query. Then the size of the complete answer aggregate for Q is of order $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$.*

Proof. In this case a subfield $Agg_y[e]$ can serve as the address of at most $h_{\mathcal{D}}$ pointers: in fact all pointers with address $Agg_y[e]$ start from some field $Agg_x[d]$ where x is the parent of y in Q and d is an ancestor of e. There are at most $h_{\mathcal{D}}$

ancestors of e, and for a fixed field $Agg_x[d]$ there is at most one pointer from $Agg_x[d]$ to $Agg_y[e]$. Hence the total number of pointers is bounded by $|Q| \cdot |D| \cdot h_{\mathcal{D}}$. It follows that the total size of Agg_Q is of order $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$. □

Examples 5.15 and 5.16 show that the bounds in Theorem 5.17 and Theorem 5.19 are sharp. It should be noted that in both bounds we could replace the size $|Q|$ of the query by the number of variables occurring in Q.

5.3 Complete answer formulae for local tree queries

So far, we have introduced complete answer aggregates for the restricted class of simple tree queries only. In this section we briefly discuss how the same concept can be used for more general classes of queries. One important characteristics of the notion of a complete answer aggregates is the principle that the administrational information that is stored in a field $Agg_x[d]$ only concerns the possible instantiations of the children of the variable x in the query Q. This restriction can be interpreted as a form of locality. Since we do not want to give up the principle, the class of Q-local constraints seems to represent a natural limit for representation techniques based on the idea of a complete answer formulae. The characterizations of complete answer formulae for local tree queries obtained in this section will be used later when treating the special case of partially ordered tree queries.

Let Q be a local tree query. Suppressing all constraints of Q that are not Q-simple we obtain a simple tree query Q_s. Let Δ_{Q_s} be the unique complete answer formula for Q_s (cf. Theorem 5.10). Each subformula $\delta_x(d)$ of Δ_{Q_s} describes the set of possible instantiations of the descendants of x under the hypothesis that x is mapped to d. These instantiations respect Q-simple constraints, but not necessarily the suppressed Q-local constraints. To circumvent this problem we add a new *restrictor condition* to each formula $\delta_x(d)$ that guarantees that the instantiation of the children $y_1, \ldots, y_h$ of x_k in Q satisfies the Q-local constraints imposed on $(x_k, y_1, \ldots, y_h)$ in Q. In principle the syntactic form of restrictor conditions is arbitrary, as long as they correctly encode Q-local constraints. For the sake of specificity we use an explicit enumeration of admissible instantiation tuples for $(y_1, \ldots, y_h)$ in the following definitions. The following definition captures the syntactical form of an appropriate class of formulae, while the notion complete answer formula defined afterwards captures the semantics.

Definition 5.20 Let $Q = (\psi \wedge c, \vec{x})$ be a local tree query, let $x \in fr(Q)$. The set of *dependent Q-instantiation formulae for x* is inductively defined as follows. Assume that $h_Q(x) = 0$. For each non-empty set $D_x \subseteq D$, the formula

$$\Delta_\epsilon^{(x)} :\equiv \bigvee_{d \in D_x} x = d$$

is a dependent Q-instantiation formula for x. Now assume that $h_Q(x) > 0$. Let $\emptyset \neq D_x \subseteq D$. Let $\{y_1, \dots, y_h\}$ denote the set of children of x in Q. For each $d \in D_x$ and each child y_i, let $\Delta_{(d)}^{(x,y_i)}$ be a dependent instantiation formula for y_i with set of target candidates $D_d^x(y_i)$. If $R_d^x(y_1, \dots, y_h)$ is a non-empty subset of $D_d^x(y_1) \times \cdots \times D_d^x(y_h)$ such that for all $i = 1, \dots, h$ and all $d_i \in D_d^x(y_i)$ there exists a tuple in $R_d^x(y_1, \dots, y_h)$ where the i-th component is d_i ("contribution obligation"), then

$$\Delta_\epsilon^{(x)} :\equiv \bigvee_{d \in D_x} (x = d \wedge (y_1, \dots, y_h) \in R_d^x(y_1, \dots, y_h) \wedge \bigwedge_{i=1}^{h} \Delta_d^{(x,y_i)})$$

is a dependent Q-instantiation formula for x. Besides the above formulae, there are no other dependent Q-instantiation formulae for x.

In the sequel, $R_d^x(y_1, \dots, y_h)$ will be called the *restrictor set* of the subformula $x = d \wedge (y_1, \dots, y_h) \in R_d^x(y_1, \dots, y_h) \wedge \bigwedge_{i=1}^{k} \Delta_d^{(x,y_i)}$. The condition that restrictor sets are always non-empty ensures that partial instantiations of dependent instantiation formulae can be extended to total instantiations (see below). The second condition on restrictor sets, which will be called "*contribution obligation*" for the sake of reference, ensures that no target candidate $d_i \in D_d^x(y_i)$ is isolated, i.e. every target candidate $d_i \in D_d^x(y_i)$ contributes to at least one answer. As in the case of simple tree queries we use expressions $\Delta_{(d_0,\dots,d_{k-1})}^{(x_0,\dots,x_k)}$ and $\delta_{(d_0,\dots,d_k)}^{(x_0,\dots,x_k)}$ for refering to subformulae of a dependent Q-instantiation formula $\Delta_\epsilon^{(x_0)}$.

Definition 5.21 Let Q be a local tree query. The set of *partial (total) instantiations* of a dependent Q-instantiation formula $\Delta_\epsilon^{x_0}$ is inductively defined as follows: for each subformula $\delta_{d_0}^{x_0}$ the mapping $\{\langle x_0, d_0 \rangle\}$ is a a partial instantiation of $\Delta_\epsilon^{x_0}$. Let $\delta_{(d_0,\dots,d_k)}^{(x_0,\dots,x_k)}$ be a subformula of $\Delta_\epsilon^{x_0}$ of the form

$$x_k = d_k \wedge (y_1, \dots, y_h) \in R_{d_k}^{x_k}(y_1, \dots, y_h) \wedge \bigwedge_{i=1}^{h} \Delta_{d_k}^{(x_k,y_i)}$$

(where $y_1, \dots, y_h$ is the sequence of children of x_k in Q). Assume that ν is a partial instantiation of $\Delta_\epsilon^{x_0}$ such that $\{\langle x_i, d_i \rangle \mid i = 1, \dots, k\} \subseteq \nu$ and ν does not instantiate any child y_i of x_k. For each tuple $(e_1, \dots, e_h) \in R_{d_k}^{x_k}(y_1, \dots, y_h)$ the mapping $\nu \cup \{\langle y_i, e_i \rangle \mid 1 \leq i \leq h\}$ is a partial instantiation of $\Delta_\epsilon^{x_0}$. There are no other partial instantiations besides those defined by the above rules. A partial instantiation ν of $\Delta_\epsilon^{x_0}$ is a (total) *instantiation* iff ν contains a pair $\langle x, d\rangle$ for every $x \in \mathit{fr}(\Delta_\epsilon^{x_0})$.

Definition 5.22 Let Q be a local tree query, with root x_0. A dependent Q-instantiation formula $\Delta_\epsilon^{(x_0)}$ for x_0 is called a *complete answer formula* for Q iff each answer to Q is a total instantiation of $\Delta_\epsilon^{(x_0)}$ and vice versa.

Theorem 5.23 *For each local tree query Q a complete answer formula Δ_Q is unique modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

The proof is a trivial variant of the proof of Theorem 5.10 and can be found in the Appendix. It can be shown that a complete answer formula for a given local tree query always exists. For the special case of ordered tree queries we shall give an algorithm for computing a complete answer formula in the next section.

Lemma 5.24 *Let Δ_Q be a complete answer formula for the local tree query Q. Then two subformulae of Δ_Q of the form $\delta_{(d_0,\ldots,d_{k-1},d_k)}^{(x_0,\ldots,x_{k-1},x_k)}$ and $\delta_{(d'_0,\ldots,d'_{k-1},d_k)}^{(x_0,\ldots,x_{k-1},x_k)}$ are always identical modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Again, the proof is a simple variant of the proof for the corresponding Lemma 5.11 and can be found in the Appendix. The crucial observation is that Lemma 4.11 holds for local queries as well. On the basis of the lemma we may write subformulae $\delta_{(d_0,\ldots,d_{k-1},d)}^{(x_0,\ldots,x_{k-1},x)}$ in the form $\delta_x(d)$ and subformulae of the form $\Delta_{(d_0,\ldots,d_{k-1},d)}^{(x_0,\ldots,x_{k-1},x,y_i)}$ in the form $\Delta_{x,y_i}(d)$.

At the end of this section we want to show that the restrictor conditions of a complete answer formula for a local tree query Q are equivalent to the Q-local (non Q-simple) constraints imposed on the respective variables in Q. A definition is needed before.

Definition 5.25 Let Q be a local tree query, let $\delta_x(d)$ be a subformula of the complete answer formula Δ_Q for Q, let $y_1, \ldots, y_h$ be the children of x. A sequence of nodes $(e_1, \ldots, e_h)$ *satisfies* a constraint $r(x, y_{i_1}, \ldots, y_{i_r})$ (where $\{y_{i_1}, \ldots, y_{i_r}\} \subseteq \{y_1, \ldots, y_h\}$) relative to d iff $r_{\mathcal{D}}(d, e_{i_1}, \ldots, e_{i_r})$ holds in $\mathcal{D}$.

Lemma 5.26 *Let $Q = (\psi \wedge c, \vec{x})$. In the situation of Definition 5.25, let R denote the restrictor set of $\delta_x(d)$, for $i = 1, \ldots, h$ let D_i be the set of target candidates* *for y_i in $\Delta_{x,y_i}(d)$. Then R is the set of all tuples $(e_1, \ldots, e_h) \in D_1 \times \cdots \times D_h$ where $(e_1, \ldots, e_h)$ satisfies all non Q-simple constraints $r(x, y_{i_1}, \ldots, y_{i_r})$ (where $\{y_{i_1}, \ldots, y_{i_r}\} \subseteq \{y_1, \ldots, y_h\}$) of c relative to d.*

Lemma 5.26, which is proven in the Appendix, shows that we may use the Q-local constraints itself as restrictor formulae. As a matter of fact this is what we expect. Though this form of representation seems natural and yields a compact representation it has the disadvantage that it might be far from obvious which tuples of target candidates for the children variables actually satisfy the relevant set of Q-local constraints. On the other hand, a naive enumeration of all elements of the restrictor set might lead to serious space problems, something that we wanted to avoid with the use of answer aggregates. Since the optimal representation of restrictor sets depends on the concrete type of Q-local constraints that are used we do not continue the discussion on this general level. Instead we treat the special case of ordered tree queries in more detail.

5.4 Complete answer aggregates for partially ordered tree queries

Partially ordered tree queries represent a special subclass of local tree queries, hence all results of the previous section can be applied.

Theorem 5.27 *For each ordered relational document structure $\mathcal{D}$ and each partially ordered tree query Q there exists a complete answer formula Δ_Q which is unique modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

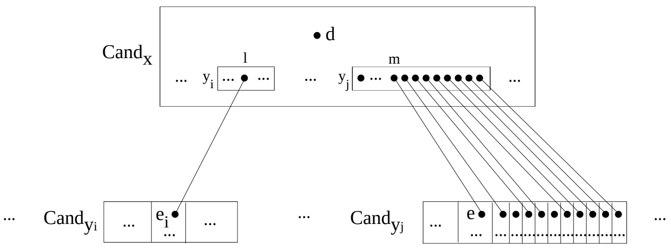
Proof. The uniqueness part is a special instance of Theorem 5.23. In Section 6 we give an algorithm that computes a complete answer formula for Q. $\square$

It remains to find a suitable representation for restrictor sets that can be used to immediately enumerate possible instantiations and leads to reasonable space requirements.

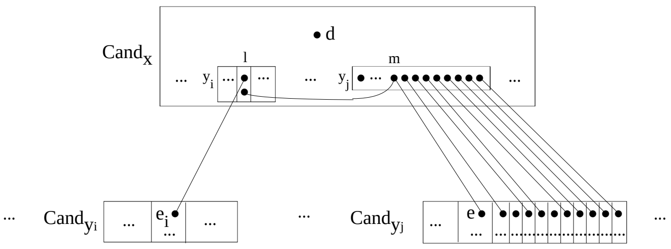
Consider a partially ordered tree query Q. Let Q_s denote the simple tree query that is obtained by suppressing all left-to-right ordering constraints and let Agg_{Q_s} be the complete answer aggregate for Q_s. We assume that the fields $Agg_y[e]$ of each record Agg_y are ordered via pre-order relation $<_{pr}^{D}$ of the nodes e. Similarly pointer lists of the form $Agg_x[d, y]$ are ordered following the ordering of their address fields.

These assumptions will help to find a simple encoding for left-to-right ordering constraints.

To illustrate the idea, consider a pointer $Agg_x[d, y_i[l]]$ of Agg_{Q_s} pointing to a field $Agg_{y_i}[e_i]$ as indicated in the figure below. Assume that Q has a constraint $y_i <_{lr} y_j$. Now let $Agg_x[d, y_j[m]]$ be the left-most pointer in $Agg_x[d, y_j]$ with an address field $Agg_{y_j}[e]$ such that $e_i <^D_{lr} e$.[4]


Cand$_x$
• d
l
m
... y_i ... • ...
... y_j • ... ••••••••••
...
... Cand$_{y_i}$ ... e_i• ...
... Cand$_{y_j}$ ... e• •••••••••••
...

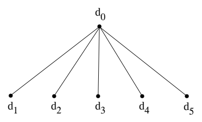
In this situation, all pointers $Agg_x[d, y_j[m']]$ with index $m' \geq m$ have address fields $Agg_{y_j}[e_{m'}]$ such that $e_i <^D_{lr} e_{m'}$, and these are the only pointers of $Agg_x[d, y_j[m']]$ with an address field satisfying this condition. In fact, by our ordering convention for fields we have $e \leq^D_p e_{m'}$ for each such m' and, since $e_i <^D_{lr} e$, Lemma 2.3 shows that $e_i <^D_{lr} e_{m'}$. This shows that all pointers have the required property. By choice of m, no other pointer can satisfy the condition. Hence, in order to encode the left-to-right ordering constraint $y_i <_{lr} y_j$ subject to the choices $x = d$ and $y_i = e_i$ it suffices to introduce a "horizontal" pointer from $Agg_x[d, y_i[l]]$ to $Agg_x[d, y_j[m]]$ as indicated in the following figure.


Cand$_x$
• d
l
m
... y_i ... • ...
... y_j • ... ••••••••••
...
... Cand$_{y_i}$ ... e_i• ...
... Cand$_{y_j}$ ... e• •••••••••••
...

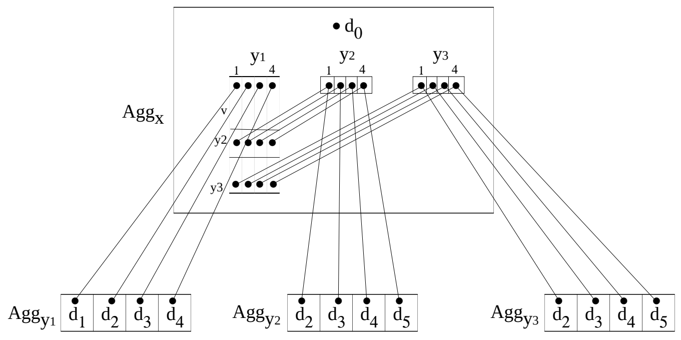
The pointer is interpreted in the following way. When instantiating x with d and y_i with e_i, we may use exactly the pointers $Agg_x[d, y_j[m]]$, $Agg_x[d, y_j[m+1]], \ldots$ for instantiating y_j. Of course, when we proceed in this way we have to introduce horizontal pointers for all possible instantiation values of variables and all left-to-right ordering constraints. We illustrate the complete picture with an example:

[4]For the sake of simplicity we assume that such a pointer exists. The discussion of the other case, where we have to erase $Agg_x[d, y_i[l]]$, is postponed to Section 6.1.

Example 5.28 Let $\mathcal{D}$ have the following form (we ignore labels and textual contents) where the left-to-right ordering between the children of d_0 is as depicted in the figure.



The complete answer aggregate for the partially ordered tree query Q of the form $(x \lhd y_1 \wedge \ldots \wedge x \lhd y_3 \wedge y_1 <_{lr} y_2 \wedge y_1 <_{lr} y_3, \langle x, y_1, y_2, y_3 \rangle)$ is the following object.



Since there are two left-to-right ordering constraints for y_1, $y_1 <_{lr} y_2$ and $y_1 <_{lr} y_3$, with each vertical pointer of $Agg[d_0, y_1]$ (line 1) we associate two horizontal pointers (lines y_2 and y_3). When instantiating y_1 with d_2, for example, we may instantiate y_2 using the pointers to d_3, d_4 or d_5, and similarly for y_3.

Definition 5.29 Let $Q = (\psi \wedge c, \vec{x})$ be a partially ordered tree query. An *aggregate* for Q is a family Agg_Q of records, $\{Agg_x \mid x \in \vec{x}\}$. Each record Agg_x is composed of an ordered sequence of subfields $Agg_x[d]$, the ordering is given by the pre-order relationship of nodes d in $\mathcal{D}$. For each child y_i of x in Q, the field $Agg_x[d]$ contains a two-dimensional array $Agg_x[d, y_i]$. With $Agg_x[d, y_i[l, *]]$ we denote the l-th column.

1. The first entry $Agg_x[d, y_i[l, v]]$ of $Agg_x[d, y_i[l, *]]$ is a "vertical" pointer, i.e., a pointer to a field of the form $Agg_{y_i}[e_i]$. Node e_i is called the *address node* of $Agg_x[d, y_i[l, *]]$. Address nodes of distinct columns are distinct.

2. For each left-to-right ordering constraint $y_i <_{lr} y_j$ of Q there is one additional entry $Agg_x[d, y_i[l, y_j]]$ in $Agg_x[d, y_i[l, *]]$ that represents a pointer to the first column $Agg_x[d, y_j[m, *]]$ with an address node e such that $e_i <_{lr}^{D} e$. There are no other entries in $Agg_x[d, y_i[l, *]]$.

Definition 5.30 Let $Q = (\psi \wedge c, \vec{x})$ be a partially ordered tree query, let Δ_Q

denote the complete answer formula for Q. A *complete answer aggregate* for Q is an aggregate $\{Agg_x \mid x \in \vec{x}\}$ for Q that satisfies the following conditions.

1. Agg_x has a subfield $Agg_x[d]$ iff Δ_Q has a subformula $\delta_x(d)$,
2. an array $Agg_x[d, y_i]$ has a vertical pointer with address field $Agg_{y_i}[e]$ iff $\delta_x(d)$ has a subformula $\delta_{y_i}(e)$.

Remark 5.31 Ignoring the trivial case of an unsatisfiable query it is again easy to see that a complete answer formula Δ_Q for a partially ordered tree query Q uniquely determines the corresponding complete answer aggregate Agg_Q. Conversely, given a complete answer aggregate Agg_Q we may reconstruct the complete answer formula Δ_Q in the following way: to obtain Δ_Q

– read the record Agg_x of the root x of Q as the disjunction of the
  formulae associated with the subfields $Agg_x[d]$,
– associate with each field $Agg_x[d]$ the conjunction of
  $x = d$ with the *horizontal pointer condition* (see below)
  and the formulae associated with the lists of pointers,
– associate with each list of pointers $Agg_x[d, y]$ of a given field $Agg_x[d]$ the
  disjunction of the formulae associated with the address fields of the pointers.

Assume that $Agg_x[d]$ has the pointer arrays $Agg_x[d, y_1], \ldots, Agg_x[d, y_h]$ for the children $y_1, \ldots, y_h$ of x in Q. The *horizontal pointer condition* has the form $(y_1, \ldots, y_h) \in R^x_d(y_1, \ldots, y_h)$ where $R^x_d(y_1, \ldots, y_h)$ contains all tuples $(e_1, \ldots, e_h)$ that satisfy the following conditions:

1. there exist pointer columns $Agg_x[d, y_1[l_1, *]], \ldots, Agg_x[d, y_h[l_h, *]]$ where vertical pointers have address nodes $e_1, \ldots, e_h$,
2. for each horizontal pointer $Agg_x[d, y_i[l_i, y_j]]$ with address $Agg_x[d, y_j[k, *]]$ we have $k \leq l_j$.

Clearly, the sets $R^x_d(y_1, \ldots, y_h)$ are exactly the restrictor sets defined in Definition 5.20. Before we show how to compute a complete answer aggregate for a partially ordered tree query we want to give an upper bound for the size.

Remark 5.32 Let $Q = (\psi \wedge c, \vec{x})$ be a partially ordered tree query, let Δ_Q denote the complete answer formula for Q. To each pointer $p = Agg_x[d, y_i[l, s]]$ of Δ_Q we assign a unique triple $(L_1(p), L_2(p), L_3(p))$ as follows.

- $L_1(p)$ is the address node of the vertical pointer $Agg_x[d, y_i[l, v]]$ of the column $Agg_x[d, y_i[l, *]]$ of p.
- We define $L_2(p) := d$. Recall that d is always an ancestor of $L_1(p)$.
- $L_3(p)$ is the following atomic subformula/constraint of Q: if $s = v$ (p is a vertical pointer) then $L_3(p) := x \lhd^{(+)} y_i$ is the formula of Q that expresses that y_i is a child of x.

  If $s = y_j$, then $L_3(p)$ is the left-to-right ordering constraint $y_i <_{lr} y_j$.

Clearly distinct pointers are mapped to distinct triples. Hence the total number of pointers of Δ_Q is bounded by the number of possible triples. This yields a bound $|D| \cdot h_{\mathcal{D}} \cdot |Q|$ for queries with soft edges, and a bound $|D| \cdot |Q|$ for rigid queries, or for labeling-complete queries over non-recursive databases. This can be seen as follows. There are $|D|$ possibilities for $L_1(p)$. $L_2(p)$ must be an ancestor (a parent for rigid queries) of $L_1(p)$. If $L_1(p)$ is fixed, there are $h_{\mathcal{D}}$ possibilities for $L_2(p)$, and just one possibility for rigid queries, or for labeling-complete queries over non-recursive databases. Since $L_3(p)$ is an atomic subformula of Q, there are $|Q|$ possibilities for $L_3(p)$.

It follows that the bounds for the size of a complete answer aggregate that we obtained for simple tree queries hold for partially ordered tree queries as well.

Theorem 5.33 *Let $\mathcal{D}$ be an ordered relational document structure and let Q be a partially ordered tree query. Then the size of the complete answer aggregate for Q is of order $\mathcal{O}(|D| \cdot h_{\mathcal{D}} \cdot |Q|)$. If Q is rigid, or if Q is labeling complete and $\mathcal{D}$ is non-recursive, then the size of the complete answer aggregate for Q is of order $\mathcal{O}(|D| \cdot |Q|)$.*

6 Computation of complete answer aggregates for tree queries

In this section we want to prove the following central results.

Theorem 6.1 *Let $\mathcal{D}$ be a relational document structure, let Q be a tree query. If Q is rigid, or if $\mathcal{D}$ is non-recursive and Q is labeling-complete, then it is possible to compute the complete answer aggregate for Q in time $\mathcal{O}(|Q| \cdot |D| \cdot log(|D|))$ and space $\mathcal{O}(|Q| \cdot |D|)$.*

Theorem 6.2 *Let $\mathcal{D}$ be an ordered relational document structure, let Q be a partially ordered tree query. Then it is possible to compute a complete answer aggregate for Q in time $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}} \cdot log(|D|))$ and space $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$.*

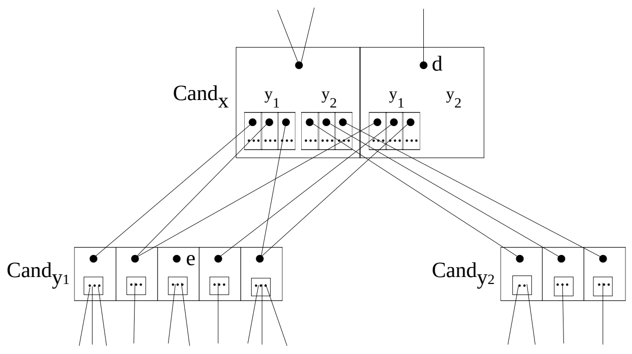
Before we describe the algorithm we characterize fields/pointer columns of aggregates that cannot contribute to any instantiation (first section). In the algorithm, these fields and pointer columns will be eliminated by a dedicated sub-procedure. The algorithm uses a special index structure for $\mathcal{D}$ that we describe in the second section. Hereafter, the algorithm itself together with its sub-procedures is described. In section four we discuss complexity issues and add an important optimization. The last part of this section contains the soundness and completeness proof.

6.1 Isolated fields and pointer columns

Given a partially ordered tree query Q, our algorithm first tries to compute a complete answer aggregate for the modified query Q_s that is obtained by suppressing all left-to-right ordering constraints of Q. In this situation the algorithm will sometimes introduce "isolated" fields that do not contribute to answers.

Definition 6.3 A field $Agg_x[d]$ of an aggregate for Q is *upwards isolated* if x is not the root of Q and if there does not exist any vertical pointer with address field $Agg_x[d]$. A field $Agg_x[d]$ is *downwards isolated* if for some child y of x the array $Agg_x[d, y]$ is empty. A field is *isolated* if it is upwards isolated or downwards isolated.

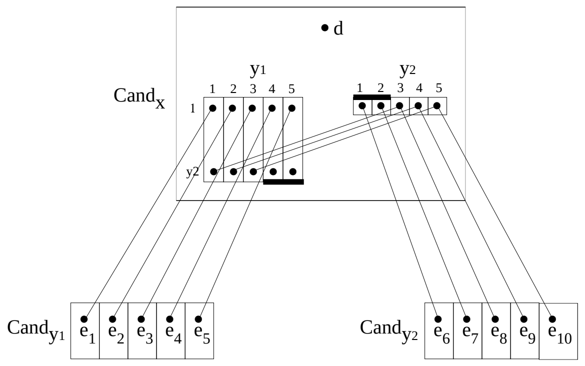
As an illustration consider the following aggregate.



Field $Agg_{y_1}[e]$ is upwards isolated. There is no value for x that would allow for an instantiation of y_1 with e. Field $Agg_x[d]$ is downwards isolated. An instantiation of x with d cannot be completed since there exists no possible instantiation for y_2 in this case. A similar problem may arise with pointer columns in connection with horizontal pointers.

Definition 6.4 A pointer column $Agg_x[d, y_i[l, *]]$ is *right isolated* if the address node of its vertical pointer $Agg_x[d, y_i[l, v]]$ is a node e_i such that for an ordering constraint $y_i <_{lr} y_j$ there is no address node e of y_j in $Agg_x[d, y_j]$ such that $e_i <_{lr}^{D} e$. A pointer column $Agg_x[d, y_j[l, *]]$ with address field $Agg_{y_j}[e_j]$ is *left isolated* if there exists a constraint $y_i <_{lr} y_j$ in Q and if the left-most horizontal pointer $Agg_x[d, y_i[1, y_j]]$ points to a column $Agg_x[d, y_j[k, *]]$ such that $k > l$.

In this situation, an instantiation of y_i (y_j) with e_i (resp. e_j) cannot contribute to a successful instantiation of Agg. As an example, consider the following record, where Q is assumed to have a constraint $y_1 <_{lr} y_2$.



Assume that $e_{10} <_{lr}^{D} e_4$. In this case an instantiation of y_1 with e_4 or e_5 cannot be completed with a suitable instantiation of y_2 such that the constraint $y_1 <_{lr} y_2$ is satisfied. The columns $Agg_x[d, y_1[4, *]]$ and $Agg_x[d, y_1[5, *]]$ are right-isolated. We say that $Agg_x[d, y_1[4, y_2]]$ and $Agg_x[d, y_1[5, y_2]]$ are "dangling" pointers with value $\perp$. Assume furthermore that e_8 is the first element e of the list $e_6, \ldots, e_{10}$ such that $e_1 <_{lr}^{D} e$. In this situation is is clear that an instantiation of y_2 with e_6 or e_7 cannot be completed with a corresponding instantiation of y_1. The pointer columns $Agg_x[d, y_2[1, *]]$ and $Agg_x[d, y_2[2, *]]$ are left-isolated.

In the description of the algorithm we shall use dangling pointers with value $\perp$. The adaption of the definition of an aggregate for Q is straightforward.

6.2 Index architecture

The algorithm for computing a complete answer aggregate for a given query uses two index structures for $\mathcal{D}$ as elaborated in this section. In order to facilitate the index access we shall assume that $\mathcal{D}$ is an *ordered* relational document structure. Even if such an ordering does not exist a priori, an artificial ordering can always be imposed on $\mathcal{D}$. This turns out to be advantageous.

In the sequel, we assume that a special finite subset K of Σ^*, called the *set of key words,* is defined. Queries are assumed to be restricted in the sense that for each formula w *in* x occurring in a query always w is a key word. Each formula of the form w *in* x ($w \in K$), $M(x)$ ($M \in \Gamma$) or $r(x)$ ($r \in \mathcal{R}$) will be called a *unary index formula*, formulae of the form $r(x,y)$ ($r \in R$) are called *binary index formulae.* When we abstract from the variables that are used in the formula we talk about (unary resp. binary) *index predicates*[5]. Note that formulae of the form $x \lhd y, x \lhd^+ y$ as well as left-to-right ordering constraints are *not* treated as index formulae. The motivation for this distinction is the following: we assume that the information that describes the tree structure of the database (i.e., the actual set of nodes, children relationship, left-to-right ordering) is separated from the index structure and stored independently.

Path selection index

The path selection index is used to retrieve for each variable that represents a leaf of the query a finite set of so-called inverted partial document paths. This step will be the basis for the algorithm to be described in the next section.

Definition 6.5 An *inverted partial document path* is a non-empty sequence of nodes $\langle d_i, d_{i-1}, \ldots, d_0 \rangle$ such that $\langle d_0, \ldots, d_i \rangle$ is a partial document path. The *initial node* of an inverted partial document path $\langle d_i, d_{i-1}, \ldots, d_0 \rangle$ is the bottom-most node d_i. An *inverted query path* has the form $\langle x_k, \ldots, x_0 \rangle$ where $\langle x_0, \ldots, x_k \rangle$ is a (complete) path of Q.

An inverted partial document path will simply be called an *inverted document path.* The path selection index contains for each unary index predicate p a list Π_p of inverted document paths.[6] Π_p is assumed to be ordered via pre-order relationship

[5]Theorems 6.2 and 6.1 and the algorithm to be described below refer to partially ordered tree queries only, therefore we can restrict ourselves to unary and binary predicates.

[6]Note that it suffices to store the initial nodes of the paths.

of initial nodes. The lists Π_p are assumed to be "sound" and "complete" in the following sense:

Remark 6.6 A node $d \in D$ is an initial node of an inverted document path in Π_p for unary index predicate p if and only if d satisfies the predicate p in $\mathcal{D}$, i.e. $\mathcal{D} \models p[d]$.

Clearly the number of inverted document paths is bounded by $|D|$. For distinct unary index predicates p and p' the intersection of the lists Π_p and $\Pi_{p'}$ can be computed in time $\mathcal{O}(|D|)$ using a simultaneous traversal of Π_p and $\Pi_{p'}$ along $<_{pr}^{D}$. This can be generalized to finite intersections.

Lemma 6.7 *Let $p_1, \ldots, p_n$ be unary index predicates. The intersection $\Pi_{p_1} \cap \ldots \cap \Pi_{p_n}$ can be computed in time $\mathcal{O}(n \cdot |D|)$.*

Given the query Q, each call to the path selection index will be triggered by an inverted query path $\langle x_k, \ldots, x_0 \rangle$. In order to simplify the presentation of the following algorithm we shall assume that the index access directly yields the intersection $\Pi_{p_1} \cap \ldots \cap \Pi_{p_n}$ where $p_1(x_k), \ldots, p_n(x_k)$ is the complete list of unary index formulae for variable x_k in Q. From Lemma 6.7 we get

Lemma 6.8 *The total time-complexity for the access to the path selection index for a query Q is bounded by $\mathcal{O}(|Q| \cdot |D|)$.*

The alignment index

The algorithm will check if an inverted document path π_D that results from the call to the path selection index for an inverted query path $\pi_Q = \langle x_k, \ldots, x_0 \rangle$ is conform with the conditions that are imposed on the variables $x_0, \ldots, x_k$ in Q. The check is organized as a bottom-up alignment process. This latter process is supported by the alignment index.

Remark 6.9 For theoretical estimation of the time-complexity of the algorithm to be described in the following section the following assumption will be made.

1. for each node d and each unary index predicate p it is possible to check in time $\mathcal{O}(log(|D|))$ if $\mathcal{D} \models p[d]$, and
2. for each pair of nodes $d_i \rightharpoonup^+ d_j$ and each binary index predicate r it is possible to check in time $\mathcal{O}(log(|D|))$ if $\mathcal{D} \models r[d_i, d_j]$.

For unary index predicates p the required test can be implemented by assigning to p a list L_p of all nodes of D that satisfy p. If L_p is ordered along the pre-order relationship of nodes and if it is accessed by binary search the requested bound is obtained.

From a theoretical point of view, a similar approach is possible for binary index predicates r, just using a list of lists, L_r. The idea is to store for each $d \in D$ the list $L_r(d)$ of all its ancestors d' such that $\mathcal{D} \models r[d, d']$. Empty lists $L_r(d)$ can be omitted. Note that the length of each list is restricted by $h_{\mathcal{D}}$. Given (d_i, d_j) we may first search for a sublist $L_r(d_i)$ of L_r, in a second step for an element d_j in $L_r(d_i)$. Both searches may be binary.

Surely this strategy is not optimal in concrete cases. However, since all binary index predicates are "generic" relations $r \in \mathcal{R}$ is seems hard to suggest a better approach that works in full generality.

Since we assume that the number of distinct index predicates is finite and constant the above assumptions lead to the following result.

Lemma 6.10 *Given a pair of document nodes (d_i, d_j) where d_j is an ancestor of d_i, and a set of index predicates, P, it is possible to check in time $\mathcal{O}(log(|D|))$ if d_j satisfies all unary predicates in P and if (d_i, d_j) satisfies all binary index predicates in P.*

6.3 The algorithm

In this section we describe an algorithm that computes a complete answer aggregate for a tree query Q. The algorithm accepts partially ordered tree queries (this includes simple tree queries) that may have rigid and soft edges. For more restricted input problems, such as rigid tree queries or simple tree queries, structural simplicity is automatically taken into account. As in the previous section we shall generally assume that on $\mathcal{D}$ a fixed pre-order relationship $<_{pr}^{D}$ is given.

In the following description of the algorithm, two phases may be distinguished. In *Phase 1*, for a given input query Q an aggregate for Q, *Agg*, is created that represents an extension of the complete answer aggregate for Q. The aggregate *Agg* may contain isolated fields and pointer columns. Two stacks will be used to collect these isolated elements. In *Phase 2*, isolated fields and pointer columns are systematically removed from *Agg* and the complete answer aggregate for Q is obtained.

Phase 1 (aggregate construction)

In this phase, two procedures are applied in iterative order. In the first step, which consists of the main procedure *INCLUDE-PATHS* with two sub-procedures *SELECT-ANCESTORS* and *CREATE*, we start with an empty Q-aggregate and enter in bottom-up manner the inverted document paths that are received from the access to the path selection index for the inverted query paths. Records, fields, and vertical pointers are created by need. In the second step, the procedure *INTR-HOR-P* computes downwards isolated fields and includes horizontal pointers for partially ordered tree queries.

```
Procedure INCLUDE-PATHS($Q, Agg, Isol\text{-}F$)
% Compute an aggregate $Agg$ for query $Q$ that is
% an extension of the complete answer aggregate for $Q$.
% Compute list $Isol\text{-}F$ of upwards isolated fields in $Agg$

  begin
    $Agg := \emptyset$;
    $Isol\text{-}F := \emptyset$;
    for each inverted path $\pi_Q := \langle x_k, \ldots, x_0 \rangle$ of $Q$        % $x_k$ leaf, $x_0$ root of $Q$
      begin
        $Agg := Agg \cup \{Agg_{x_k}\}$ where $Agg_{x_k}$ is an empty record
                marked as new;   % ($\dagger_1$)
        $\Pi$ := set of inverted document paths obtained from access to path selection
                index for $\pi_Q$;   % ($\dagger_2$)
        for each path $\pi = \langle d_m, d_{m-1}, \ldots, d_0 \rangle$ of $\Pi$        % $d_0$ topmost node
          begin
            introduce a field $Agg_{x_k}[d_m]$;
            apply SELECT-ANCESTORS($Agg, x_k, d_m, \langle d_m, d_{m-1} \ldots, d_0 \rangle, Isol\text{-}F$);
          end
        if one of the new records $Agg_{x_k}$ is empty then
          fail;        % $\dagger_3$ no document path matches $\Rightarrow$ no solution
        else
          mark all new records as old;   % $\dagger_4$ preparation for next query path
      end
    return $Agg$;
    return $Isol\text{-}F$;
  end;
```

```
Procedure SELECT-ANCESTORS($Agg, y, d_i, \langle d_i, \ldots, d_0 \rangle, Isol\text{-}F$)
% Enter the inverted partial document path $\langle d_i, \ldots, d_0 \rangle$
% into $Agg$ (if possible), begin with $d_i$ in $y$.
% Update list of isolated fields Isol-F.
begin
        if ($y$ = root of $Q$ or $i = 0$) then
            begin
                if $y$ is not root of $Q$ then       % $\dagger_5$, $Agg_y[d_i]$ upwards isolated
                    add $Agg_y[d_i]$ to Isol-F;
                else stop;                 % alignment of the path is finished successfully
            end;
        else begin
            $x$ := parent of $y$ in $Q$;
            if $y$ is a rigid child of $x$ in $Q$ then
                begin
                    if $\langle d_i, d_{i-1} \rangle$ ($d_i$) satisfy binary (unary) index formulae for
                                            $\langle y, x \rangle$ ($x$) in $Q$ then
                        apply CREATE($x, d_{i-1}, Agg, y, d_i, \langle d_i, \ldots, d_0 \rangle, Isol\text{-}F$);
                    else add $Agg_y[d_i]$ to Isol-F;
                end
            else                           % $y$ is a soft child of $x$ in $Q$
                begin
                    Node-Found := false;
                    for each node $d_j$ in $\{d_{i-1}, \ldots, d_0\}$
                        if $\langle d_i, d_j \rangle$ ($d_i$) satisfy binary (unary) index formulae
                                            for $\langle y, x \rangle$ ($x$) in $Q$ then
                            begin
                                Node-Found := true;
                                apply CREATE($x, d_j, Agg, y, d_i, \langle d_i, \ldots, d_0 \rangle, Isol\text{-}F$);
                            end
                    if not Node-Found then
                        add $Agg_y[d_i]$ to Isol-F;
                end;
        end;
    end;
```

```
Procedure CREATE($x, d_j, Agg, y, d_i, \langle d_i, \ldots, d_0 \rangle, Isol\text{-}F$)
% Add node $d_j$ to field $Agg_x$ (if possible) with pointers to node
% $d_i$ in $Agg_y$. If $d_j$ did not already exist, continue with
% remaining nodes in $\langle d_j, \ldots, d_0\rangle$.
% Update list of isolated fields Isol-F.
begin
    if record $Agg_x$ does not exist then
      begin
        introduce empty record $Agg_x$ marked as new;
        introduce $Agg_x[d_j]$ with empty pointer arrays
            $Agg_x[d_j, z]$ for the children $z$ of $x$ in $Q$;
        add pointer column $Agg_x[d_j, y[1,*]]$ to $Agg_y[d_j, y]$;
        introduce a vertical pointer from $Agg_x[d_j, y[1,v]]$ to $Agg_y[d_i]$;
        apply SELECT-ANCESTORS$(Agg, x, d_j, \langle d_j \ldots, d_0\rangle)$;
      end
    else if $Agg_x$ exists and is marked old then
      begin
        if field $Agg_x[d_j]$ exists then
          begin
            add a new pointer column $Agg_x[d_j, y[k,*]]$ to $Agg_x[d_j, y]$;
              % at this step the correct ordering of pointer columns
              % mirroring the order of address nodes has to be respected
            introduce a vertical pointer from $Agg_x[d_j, y[k,v]]$ to $Agg_y[d_i]$;
                          % †6
          end
        else add $Agg_y[d_i]$ to ISOL-F;     %†7,   $Agg_y[d_i]$ upwards isolated
      end
    else if $Agg_x$ exists and is marked new then
      begin
        if field $Agg_x[d_j]$ exists then
          begin
            add a new pointer column $Agg_x[d_j, y[k,*]]$ to $Agg_x[d_j, y]$;
              % at this step the correct ordering of pointer columns
              % mirroring the order of address nodes has to be respected
            introduce a vertical pointer from $Agg_x[d_j, y[k,v]]$ to $Agg_y[d_i]$;
                          % †8
          end
        else                  % $Agg_x[d]$ does yet not exist
          begin
```

```
                introduce a field $Agg_x[d_j]$ with empty pointer lists
                    $Agg_x[d_j, z]$ for the children $z$ of $x$ in $Q$;
                  % the correct ordering of fields has to be respected
                add a pointer column $Agg_x[d_j, y[1, *]]$ to $Agg_x[d_j, y]$;
                introduce a vertical pointer from $Agg_x[d_j, y[1, v]]$ to $Agg_y[d_i]$;
                apply $SELECT\text{-}ANCESTORS(Agg, x, d_j, \langle d_j \ldots, d_0 \rangle)$;
            **end**;
        **end**;
    **end**;
```

Let us summarize the procedures. Given the query Q, inverted query paths are treated in consecutive order. For each inverted query path π_Q with initial node x_k, the access to the path selection index yields a set of inverted document paths, Π ($\dagger_2$). If P denotes the set of all index formulae for x_k in Q, the set Π represents the intersection of all the sets of document paths that are associated with predicates $p \in P$ in the path selection index, as described in 6.2.[7]
Given an inverted path $\pi \in \Pi$, the inverted query path π_Q is recursively matched against π in a bottom-up manner (i.e., from the leaf to the root of Q), checking at each step if the relevant index formulae are satisfied. The starting point for each single matching step is a situation where we successfully aligned two prefixes of the inverted paths π and π_Q ending at nodes d_i and y respectively. If we have reached the root either of $\mathcal{D}$ or of Q the process stops (in such a situation, if we have *not* reached the root of Q, the actual field is upwards isolated and moved to the stack of isolated fields, $\dagger_5$). Otherwise the possible choices for the next matching step depend on whether y is a rigid child or a soft child of its parent node x in Q. In the former case we obtain (at most) one subcall to *CREATE*, where we climb from the aligned children nodes y and d_i to the parent nodes in π_Q and π respectively. For soft edges, the parent node x of the query paths can possibly be aligned with several ancestor nodes d_j of d_i, and for each such ancestor we have one call to *CREATE*. The subprocedure *CREATE* builds up the aggregate, introducing new arrays, fields, and vertical pointers by need. We shall assume that vertical pointers are always bi-directional. This will simplify the elimination of isolated pointer columns in Step 2.

A remarkable feature of the algorithm, located in subprocedure *CREATE* (cf.

[7]If P is empty, then Π is the set of all inverted paths in the document database. Note that these pathological cases do not destroy the worst-case time complexity of the algorithm.

$\dagger_6$ and $\dagger_8$), is the following. Whenever we encounter a field $Agg_x[d]$ that already exists, after introducing an appropriate pointer we stop. The intuitive justification is that all the structure (pointers and fields) that we would obtain by continuation of the alignment process has already been included in the aggregate. This follows from the fact that prefixes of (top-down) document paths that end at the same node must necessarily be identical. Hence, when meeting an existing field $Agg_x[d]$, the prefix of the actual document path that has still to be consumed is identical to the prefix of another path that had been aligned earlier, starting from the same field of the aggregate. Since the result will necessarily be the same, we do not have to do the work twice. Formally it follows that the subprocedure *SELECT-ANCESTORS* is never called with the same pair (y, d_i) twice, which is important for the estimate of the worst-case complexity of the algorithm that we shall give below.

A record Agg_x is marked "old" (resp. "new") if it has (resp. has not) been created before the treatment of the actual query path π_Q ($\dagger_1, \dagger_4$). A particular situation arises when we try to enter a new field into an old record in the alignment process ($\dagger_7$). It is simple to see that such a field cannot contribute to any instantiation of the aggregate since it cannot be combined with the document paths that we introduced when treating the previous query paths. In other words, such a field would necessarily be downwards isolated. For this reason it is not introduced, which means that the previously visited field is upwards isolated.

If at the end of the treatment of a query path π_Q one of the new records Agg_{x_k} is empty after all document paths $\pi \in \Pi$ have been considered ($\dagger_3$), then the aggregate cannot be instantiated, i.e., the query has no answer, regardless of possible future steps. We stop with failure in such a situation (we might output x_k to the user).

We should add a general remark on the use of stacks for isolated elements (i.e., fields or columns). When adding (a pointer to) an element to one of the two stacks, we colour the element in the aggregate using a marker "yellow" that indicates that the element is on the stack. Whenever we say that an element is added to the stack we always mean that we first check if the element is marked "yellow". In this case the element is *not* added again to the stack.

We now describe the subprocedure that introduces horizontal pointers. Since we have to visit each pointer array the procedure is also used to compute the set of downwards isolated fields of Agg. The remaining subpart of the procedure ($\dagger_9$) becomes vacuous in cases where Q does not have ordering constraints. The procedure represents the second and final step of Phase 1.

```
Procedure INTR-HOR-P($Q, Agg, Isol\text{-}F, Isol\text{-}PC$)
% Compute for query $Q$ and aggregate $Agg$ set of isolated fields, isolated
% pointer columns and introduce horizontal pointers into $Agg$.
begin
    Isol-PC:= empty stack;
    for each array $Agg_x[d, y_i]$ of $Agg$;
      begin
        if $Agg_x[d, y_i]$ = empty array then
          add $Agg_x[d]$ to Isol-F;                        % field downwards isolated
        else
          for each constraint $y_i <_{lr} y_j$ of $Q$      %    $\dagger_9$
            begin
              for each column $Agg_x[d, y_i[l, *]]$ of $Agg_x[d, y_i]$
                begin
                  $e_i$ := address node of $Agg_x[d, y_i[l, v]]$;
                  if exists $k$ := minimal number s.th. address node $e$ of
                          $Agg_x[d, y_j[k, v]]$ satisfies $e_i <^D_{lr} e$ then
                    begin
                      if $l = 1$ and $k > 1$ then              % left isolated columns
                        for all $k' < k$ add $Agg_x[d, y_j[k', *]]$, to Isol-PC;
                      introduce hor. pointer $Agg_x[d, y_i[1, y_j]]$ with address $Agg_x[d, y_j[k, *]]$;
                    end
                  else
                    begin
                      set pointer $Agg_x[d, y_i[l, y_j]]$ to $\perp$;
                      add $Agg_x[d, y_i[l, *]]$ to Isol-PC;     % right isolation
                    end
                begin
            end
      end
    return Agg;
    return Isol-F;
    return Isol-PC;
  end;
```

The procedure has as input the query Q, the aggregate *Agg* and the stack of isolated fields *Isol-F* computed in the first step. It visits each array $Agg_x[d, y_i]$ of the aggregate. If an array is empty, the corresponding field is downwards isolated and added to the stack of isolated fields. In the other case, all relevant left-to-right ordering constraints $y_i <_{lr} y_j$ are considered. For each pointer column the procedure tries to introduce the appropriate horizontal pointer. If this is not possible (i.e., for right isolated pointer columns) a "dangling" pointer is introduced and the pointer column is added to the stack of isolated pointer columns. When treating the first pointer column $Agg_x[d, y_i[1, *]]$ we also check if $Agg_x[d, y_j]$ contains left isolated pointer columns. These are added to the stack of isolated pointer columns.

Phase 2 (elimination of isolated fields/pointer columns)

The input for Phase 2 is the aggregate *Agg* together with the stack of isolated fields *Isol-F* and the stack of isolated pointer columns *Isol-PC* as computed in Phase 1. In the aggregate, the elements of these stacks are marked "yellow".

Basically the following procedure is very simple. We take the eliminable elements from the stacks and erase them. Since the erasure of an isolated element may lead to new isolated elements the process has to be organized in a recursive way. If during this process a record Agg_x becomes empty, then the procedure stops (*FAIL* = true). Otherwise it continues until all isolated fields and pointer columns are erased.

In the presence of left-to-right ordering constraints, the strategy has to be modified. We do not immediately erase an isolated pointer column that serves as the address of horizontal pointers. The reason is that we have to re-address these horizontal pointers, using the column that represents the right neighbour as the new address. With the naive strategy this process possibly would have to be iterated, we would end up with a quadratic complexity.

Hence, when treating yellow pointer columns that serve as the target of horizontal pointers we proceed in two steps. Instead of erasing the column, we only eliminate the (vertical and horizontal) pointers departing from the column and colour the column "red" afterwards. Horizontal pointers to red columns are only re-addressed once, after all yellow elements have been treated (eliminated or coloured).

```
    Procedure $CLEAN(Agg, Isol\text{-}PC, Isol\text{-}F)$
% remove isolated fields and pointer columns in $Agg$, triggering
% new isolated fields and pointer columns.
```

```
begin
    $\mathit{FAIL} :=$ false;
    until $\mathit{FAIL}$ or $\mathit{Isol\text{-}PC} = \mathit{Isol\text{-}F} = \emptyset$;
        begin
            if $\mathit{Isol\text{-}F} \neq \emptyset$ then
                begin
                    $\mathit{Agg}_x[d] :=$ pop($\mathit{Isol\text{-}F}$);
                    apply $\mathit{ELIM\text{-}F}(\mathit{Agg}, x, d, \mathit{Isol\text{-}PC}, \mathit{Isol\text{-}F}, \mathit{FAIL})$;
                end
            else
                begin
                    $\mathit{Agg}_x[d, y[l, *]] :=$ pop($\mathit{Isol\text{-}PC}$);
                    apply $\mathit{ELIM\text{-}PC}(\mathit{Agg}, x, d, y, l, \mathit{Isol\text{-}PC}, \mathit{Isol\text{-}F})$;
                end
        end
    if $\mathit{FAIL}$ then fail;        % query has no answer
    for each red pointer column $\mathit{Agg}_x[d, y[l, *]]$ of $\mathit{Agg}$
        begin
            select minimal $l' > l$ where column $\mathit{Agg}_x[d, y[l', *]]$ not red;
            re-address all horizontal pointers with target $\mathit{Agg}_x[d, y[l, *]]$
                    using new address $\mathit{Agg}_x[d, y[l', *]]$;
            erase $\mathit{Agg}_x[d, y[l, *]]$;
        end;
  end;
```

The following subprocedure eliminates/colours isolated pointer columns. We say that the address field $\mathit{Agg}_y[e]$ of a vertical pointer $\mathit{Agg}_x[d, y[l, v]]$ is *upwards isolated up to* $\mathit{Agg}_x[d, y[l, v]]$ iff $\mathit{Agg}_x[d, y[l, v]]$ is the only vertical pointer with address field $\mathit{Agg}_y[e]$.

```
Procedure $\mathit{ELIM\text{-}PC}(\mathit{Agg}, x, d, y, l, \mathit{Isol\text{-}PC}, \mathit{Isol\text{-}F})$
% eliminate column $y[l, *]$ in field $\mathit{Agg}_x[d]$
% new isolated fields and pointer columns can be created and
% are added to $\mathit{Isol\text{-}F}$ and $\mathit{Isol\text{-}PC}$.

begin
```

```
if $Agg_x[d, y[l, v]]$ not dangling then
  begin
    $Agg_y[e]$ := address field of $Agg_x[d, y[l, v]]$;
    if $Agg_y[e]$ is upwards isolated up to $Agg_x[d, y[l, v]]$ then
      add $Agg_y[e]$ to Isol-F;
  end
erase $Agg_x[d, y[l, v]]$;


if $Agg_x[d, y[l, *]]$ is the only non-red column of $Agg_x[d, y]$ then
    add $Agg_x[d]$ to Isol-F;          % †10


else
  begin
    if $Agg_x[d, y[l, *]]$ leftmost non-red column of $Agg_x[d, y]$ then
      begin
        let $Agg_x[d, y[l', *]]$ be the next non-red column of $Agg_x[d, y]$;
        for each hor. pointer $Agg_x[d, y[l', y_i]]$ of $Agg_x[d, y[l', *]]$     % †11
          begin
            let $Agg_x[d, y_i[k, *]]$ := target of $Agg_x[d, y[l', y_i]]$;
            for each $k' < k$ where column $Agg_x[d, y_i[k', *]]$ neither yellow nor
                    red add $Agg_x[d, y_i[k', *]]$ to Isol-PC;   % †12
          end
      end


    else if $Agg_x[d, y[l, *]]$ right-most non-red column of $Agg_x[d, y]$ then
      begin
        let $Agg_x[d, y[l_0, *]]$ be the largest non-red column smaller than $Agg_x[d, y[l, *]]$;
        let $l_1, \ldots, l_k = l$ denote the indexes of successor columns of
              $Agg_x[d, y[l_0, *]]$ ending at $Agg_x[d, y[l, *]]$;
        for each column $Agg_x[d, y[l_i, *]]$ $(1 \leq i \leq k)$
          for each horizontal pointer $Agg_x[d, y_j[m, y]]$ with address $Agg_x[d, y[l_i, *]]$
            add $Agg_x[d, y_j[m, *]]$ to Isol-PC;   % †13
      end


    erase all horizontal pointers departing from $Agg_x[d, y[l, *]]$;
    if $Agg_x[d, y[l, *]]$ is not target of any horizontal pointer then
      erase $Agg_x[d, y[l, *]]$;
```

```
            else colour $Agg_x[d, y[l, *]]$ red;
        end;
    end;
```

During a call to *ELIM-PC* the vertical pointer of the column is erased, adding the target field to *Isol-F* if the field becomes upwards isolated. When treating the last non-red column of an array we know that after the final removal of red columns the actual field will be downwards isolated. Hence the field is added to *Isol-F* (cf. $\dagger_{10}$). In the situation of $\dagger_{12}$ the columns $Agg_x[d, y_i[k', *]]$ are necessarily left-isolated when removing red columns. In the situation of $\dagger_{13}$ the columns $Agg_x[d, y_i[k', *]]$ are necessarily right-isolated when removing red columns.

It remains to describe the procedure that eliminates isolated fields.

```
Procedure ELIM-F($Agg, x, d, Isol\text{-}PC, Isol\text{-}F, FAIL$)
% eliminate isolated field $Agg_c[d]$
% if record $Agg_x$ becomes empty, the flag $FAIL$ is set to true

begin
    for each vertical pointer $Agg_z[d', x[k, v]]$ with address field $Agg_x[d]$
        begin
            redefine address of $Agg_z[d', x[k, v]] := \perp$;
            add $Agg_z[d', x[k, *]]$ to Isol-PC;
        end

    for each child $y$ of $x$ in $Q$
        for each column $Agg_x[d, y[l, *]]$ of $Agg_x[d, y]$
            begin
                $Agg_y[e]$ := address field of $Agg_x[d, y[l, v]]$;
                if $Agg_y[e]$ is upwards isolated up to $Agg_x[d, y[l, v]]$ then
                    add $Agg_y[e]$ to Isol-F;
                erase $Agg_x[d, y[l, *]]$
            end

    erase $Agg_x[d]$;
    if $Agg_x$ = empty record then
        $FAIL$ := true;
```

```
  end;
```

6.4 Complexity and practical optimization

Before we state the main complexity result we start with some general remarks.

Remark 6.11 The pre-order relationship $<_{pr}^{D}$ on $\mathcal{D}$ can be represented by assigning to every node d a natural number $\mathrm{ord}(d)$ as identifier so that

$$\mathrm{ord}(d_1) < \mathrm{ord}(d_2) \text{ iff } d_1 <_{pr} d_2.$$

We assume that the comparison of two natural numbers is of constant-time complexity.[8] Procedure *INTR-HOR-P* also includes tests $d_1 <_{lr}^{D} d_2$. For these tests we use a supplementary pointer structure: each node $d \in D$ has a pointer to the node $e = \mathit{minsucc}\ (d)$ that represents the first successor of d with respect to $<_{pr}^{D}$ that is larger than d with respect to left-to-right ordering. In other words, we have $d <_{lr}^{D} e$ and there is no node $e' <_{pr}^{D} e$ such that $d <_{lr}^{D} e'$. With this prerequisite, Lemma 2.3 allows to reduce the left-to-right ordering to the pre-order relationship: in fact the lemma shows that for $d_1, d_2 \in D$ we have $d_1 <_{lr}^{D} d_2$ iff $\mathit{minsucc}\ (d_1) \leq_{p}^{D} d_2$ and the latter formula can be tested in constant time.

Remark 6.12 Let $E \subseteq D$ and assume that the pre-order relationship "$<_{pr}^{D}$" on E is encoded using a binary tree with height $\log(|D|)$ where the elements of E are represented by the leaves. Assume that each node is coloured black, yellow, or red. Then the following operations can be computed in time $\mathcal{O}(\log(|D|))$:

- find the first predecessor (successor) (w.r.t. "$<_{pr}^{D}$") of a given element that has a given colour,
- check if a given element is the only (right-most, left-most) element of a given colour,
- change the colour of a given element.

[8]In fact, if we do not impose an upper bound on the natural numbers, the time complexity of the comparison is $\mathcal{O}(\log(n))$. But it is a reasonable and canonical assumption in database theory to impose an upper bound on the size of databases, e.g. that the database should not contain more than 2^{32} nodes. Then we can rely on efficient hardware treatment for the comparison of two 32-bit integers.

The operations can be implemented by adding to each inner node of the binary tree a label "black" ("yellow", or "red") iff it has a successor leaf that has the respective colour.

We may now state the main complexity result.

Theorem 6.13 *The worst-case time-complexity of the algorithm described in Section 6.3 is of order $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}} \cdot log|D|)$. If Q is rigid, or if $\mathcal{D}$ is non-recursive and Q is labeling-complete, then the time-complexity is $\mathcal{O}(|Q| \cdot |D| \cdot log(|D|))$.*

Proof. Lemma 6.8 shows that we may ignore the access to the path selection index for obtaining the above bound.

First we consider *Step 1 of Phase 1* of the algorithm.
As we noted in the description of the algorithm the total number of calls to procedure *SELECT-ANCESTORS* is bounded by $|Q| \cdot |D|$. Each non-trivial test in *SELECT-ANCESTORS* needs time bounded by $\mathcal{O}(log(|D|))$ (cf. Remark 6.9), the number of tests in one call to *SELECT-ANCESTORS*, similarly as the number of possible calls to *CREATE*, is bounded by $h_{\mathcal{D}}$. Let us investigate each of the steps of procedure *CREATE*. Obviously, given $x \in fr(Q)$ it is possible (e.g., by adding appropriate information to the query) to check in constant-time if a record Agg_x exists and to determine whether it is marked as new or old. Using binary search it takes time $\mathcal{O}(log(|D|))$ to check if a field $Agg_x[d_j]$ exists for given x and d_j. The same bound holds for the introduction of new pointer columns and fields where pre-ordering has to be respected. It follows that one call to *CREATE* needs time $\mathcal{O}(log(|D|))$. Hence we obtain a bound $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}} \cdot log(|D|))$ for the first step.

We consider *Step* 2 *of Phase 1* of the algorithm (procedure *INTR-HOR-P*). Remark 5.32 shows that the total number of horizontal pointers is bounded by $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$. Using binary search it takes time $\mathcal{O}(log(|D|))$ to determine the correct address for a given pointer. Since each column is added to the stack of isolated columns at most once we receive the bound $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}} \cdot log(|D|))$ for Step 2. Summing up, this bound is also obtained for Phase 1.

If Q is rigid, or if Q is labeling-complete and $\mathcal{D}$ is non-recursive, then each call to *SELECT-ANCESTORS* leads to just one test and to at most to one call of *CREATE*. We obtain a bound $\mathcal{O}(|Q| \cdot |D| \cdot log(|D|))$ for the first step. The number of pointer columns/vertical pointers that are introduced in Phase 1 is bounded by $|Q| \cdot |D|$. By Remark 5.32, the total number of pointers and pointer columns is bounded by $\mathcal{O}(|Q| \cdot |D|)$ and we receive the bound $\mathcal{O}(|Q| \cdot |D| \cdot log(|D|))$ for Phase 1.

A simple analysis of *Phase 2* shows that for each element (field/pointer/pointer column) there is only a fixed number of operations that is possibly applied (put the element on a stack, colour it, compute/redefine address, erase the element, check if element is only non-red column, or left-most non-red column etc.). Each operation is applied at most once to a given element. This is fairly obvious, we just add some remarks. The horizontal pointers that are inspected at $\dagger_{11}$ are only inspected once. In fact these pointers belong to the second non-red column of a given array; after finishing the actual call to *ELIM-PC* this column will be the first column of the array, which shows that the same pointers cannot be inspected a second time at $\dagger_{11}$. At $\dagger_{12}$ note that the computation of each column that has to be added to *Isol-PC* takes time $\mathcal{O}(\log(|D|))$, by Remark 6.12. The same remark shows that each of the operations mentioned above can be applied in time $\mathcal{O}(\log(|D|))$. Since the number of elements is bounded by $\mathcal{O}(|Q| \cdot |D| \cdot h_{\mathcal{D}})$ (general case) and $\mathcal{O}(|Q| \cdot |D|)$ (rigid queries, or labeling-complete queries over non-recursive databases) respectively, we receive the desired complexity bounds for Phase 2. □

In [Meu98] an even better result for rigid queries was obtained by treating all paths in a path set returned by the index simultaneously.

For practical purposes the following modification of the algorithm seems to be highly preferable. For each query path π_Q, we first determine the cardinality $n(\pi_Q)$ of the set of inverted document paths $\Pi(\pi_Q)$ that is obtained from access to the path selection index for π_Q. The algorithm then treats query paths in the order determined by the numbers $n(\pi_Q)$, starting with the smallest numbers. When treating a new query path π_Q, the bottom-most variable x of π_Q that has occurred already in one of the earlier query paths is computed. Here Agg_x is the first record with marker "old" that is reached when we enter in bottom-up manner the document paths obtained from the access to the path selection index into the aggregate. An inverse document path $\pi_D \in \Pi(\pi_Q)$ can only be entered successfully if it contains a node d such that Agg_x has a field $Agg_x[d]$. Let us call such a path *relevant.* We show how to compute the subset of relevant paths of $\Pi(\pi_Q)$ in linear time.

Let $\langle d_1, \ldots, d_m \rangle$ denote the sequence of nodes of the record Agg_x, in pre-order enumeration. An element $d \in \{d_1, \ldots, d_m\}$ is called *maximal* iff no ancestor of d belongs to $\{d_1, \ldots, d_m\}$. For each node $d_l \in \{d_1, \ldots, d_m\}$, the set of descendants defines an interval of the form $[d_l, d_l^*]$ of the pre-order relation on $\mathcal{D}$. Node d_l^* is the right-most leaf among the descendants of d_l and can be obtained in constant time (cf. Remark 6.11) using the formula $d^* = pred_p(minsucc\ (d))$. The expression $pred_p(e)$ stands for the predecessor of e with respect to pre-order relation $<_{pr}$. Two intervals $[d_l, d_l^*]$ and $[d_h, d_h^*]$ are either disjoint or one is contained in the other. The intervals of maximal elements are pairwise disjoint and cover all the intervals of nodes in $\{d_1, \ldots, d_m\}$. The sequence of all maximal elements d_l, together with the right boundaries d_l^* of their intervals, can be computed in time $\mathcal{O}(m)$. In fact, d_1 is always maximal. Once we have found that d_l is maximal, the first element of $d_{l+1}, \ldots, d_m$ that does not belong to $[d_l, d_l^*]$ is the next maximal element of the sequence.

Obviously a path $\pi_D \in \Pi(\pi_Q)$ with initial node e is relevant iff e is in the pre-order interval $[d_l, d_l^*]$ for a maximal node d_l of $\{d_1, \ldots, d_m\}$. Recall that each set $\Pi(\pi_Q)$ is ordered (via index access) according to the pre-order of the initial (bottom-most) nodes. Let $\langle e_1, \ldots, e_n \rangle$ denote the sequence of all initial nodes of paths in $\Pi(\pi_Q)$, ordered in this way. Using one simultaneous traversal of (the intervals associated with) the subsequence of maximal nodes in $\langle d_1, \ldots, d_m \rangle$ on one hand and $\langle e_1, \ldots, e_n \rangle$ on the other hand we may filter out the list of all relevant paths in time $\mathcal{O}(n + m) \leq \mathcal{O}(|D|)$.

Since the number of query paths is bounded by $|Q|$ the total time-complexity of all filtering steps is bounded by $\mathcal{O}(|Q| \cdot |D|)$. This shows that the bound for the worst-case complexity of the algorithm is not affected by this filtering.

6.5 Soundness and completeness

In this section we prove that the given algorithm in fact computes the complete answer aggregate for the input query, which completes the proof of Theorems 6.1 and 6.2. In view of the bidirectional translation between complete answer formulae and complete answer aggregates (cf. Remarks 5.14 and 5.31) this also proves the existence parts of Theorems 5.10 and 5.27. Clearly simple tree queries are a special case of partially ordered tree queries, hence we may restrict considerations to the latter type of queries. Since the algorithm computes complete answer aggregates as opposed to complete answer formulae we first give an internal characterization of complete answer aggregates. We will show that our algorithm computes an aggregate that satisfies the conditions of this characterization.

Definition 6.14 A mapping $\nu : fr(Q) \to D$ is an *instantiation* of the aggregate

$Agg_Q = \{Agg_x \mid x \in fr(Q)\}$ iff the following conditions are satisfied:

1. Agg_x has a field $Agg_x[\nu(x)]$ for all $x \in fr(Q)$,
2. if y is a child of x in Q, if $d = \nu(x)$ and $e = \nu(y)$, then e is an address node of a vertical pointer of a column $Agg_x[d, y[l, *]]$ of the array $Agg_x[d, y]$,
3. if $\nu(x) = d$ and Agg_Q has a horizontal pointer $Agg_x[d, y_i[l, y_j]]$ with address $Agg_x[d, y_j[m, *]]$, if $e_i := \nu(y_i)$ is the address node of the vertical pointer $Agg_x[d, y_i[l, v]]$, then $e_j := \nu(y_j)$ is the address node of a vertical pointer $Agg_x[d, y_j[m', v]]$ such that $m' \geq m$.

We say that each field $Agg_x[\nu(x)]$ *belongs to* the instantiation ν. Similarly each pointer column $Agg_x[d, y[l, *]]$ of the form described in 2 is said to belong to ν.

We may now give the internal characterization of the complete answer aggregate for Q.

Lemma 6.15 *Let Q be a partially ordered tree query. An aggregate Agg is the complete answer aggregate Agg_Q for Q iff the following conditions are satisfied:*

1. *Each instantiation ν of Agg is an answer to Q and vice versa,*
2. *every field/pointer column of Agg belongs to an instantiation of Agg.*

The simple proof is omitted. We note that Condition 2 exactly corresponds to the "contribution obligation" condition for dependent Q-instantiation formulae (cf. paragraph below Definition 5.20). Hence it remains to prove that the aggregate *Agg* that represents the output of the algorithm satisfies Conditions 1 and 2. Let us start with some simple observations.

Lemma 6.16 *Let Agg_1 and Isol-F_1 denote the output of INCLUDE-PATHS. Then all upwards isolated fields of Agg_1 are in Isol-F_1.*

Proof. This follows from the fact that whenever we cannot finish the bottom-up alignment of inverted query path and inverted document paths we add the field of the last successful alignment step to *Isol-F_1* (cf. *SELECT-ANCESTORS* and *CREATE*). □

Lemma 6.17 *Let Agg_2, Isol-F_2 and Isol-PC_2 denote the output of Phase 1 (i.e., the output of INTR-HOR-P). Then all isolated fields and pointer columns of Agg_2 are in Isol-F_2 and Isol-PC_2 respectively.*

Proof. Clearly Lemma 6.16 implies that upwards isolated fields are in *Isol-F_2*. *INTR-HOR-P* treats each array $Agg_x[d, y]$ of the aggregate and adds downwards isolated fields to *Isol-F*. It also adds each right or left isolated pointer column to *Isol-PC*. □

During Phase 2, let us call a field/pointer column *quasi-isolated* if the element becomes isolated when removing red pointer columns.

Lemma 6.18 *At each time of the computation in Phase 2, each quasi-isolated field is either on the actual stack Isol-F or it represents the actual argument of the elimination sub-procedure that is executed. Each quasi-isolated pointer column is either on the actual stack Isol-PC, or it represents the actual argument of the elimination sub-procedure that is executed, or it is coloured "red".*

Proof. Follows from Lemma 6.17 by a trivial induction and inspection of *CLEAN*, noticing that a new quasi-isolated field/pointer column can only be the result of the elimination/red colouring of a field/pointer column that had been quasi-isolated previously. □

Lemma 6.19 *Let Agg_3 denote the output aggregate of procedure CLEAN (i.e., the output of the algorithm). Then Agg_3 does not have isolated fields or pointer columns.*

Proof. Consider the situation in *CLEAN* where both *Isol-F* and *Isol-PC* are empty. In this situation by Lemma 6.18, the only quasi-isolated elements that are left are the red pointer columns. These columns are erased in *CLEAN*. Clearly the elimination of a red column cannot lead to a new quasi-isolated element. □

Lemma 6.20 *If an aggregate Agg for Q does not have any isolated field/pointer column, then every field/pointer column of Agg belongs to an instantiation of Agg.*

Proof. We proceed by induction on $h_Q(x)$ where x is the root of Q. If $h_Q(x) = 0$, then Agg_x is the only record of *Agg* and the statement is trivial. Assume now that

$h_Q(x) > 0$, let $y_1, \dots, y_h$ denote the children of x in Q. Let Agg_i denote the sub-aggregate with topmost record Agg_{y_i} $(1 \leq i \leq h)$. By induction hypothesis each field/pointer column of Agg_i belongs to an instantiation of Agg_i. Now let $Agg_z[d_0]$ be a field of Agg. We distinguish two cases.

In the first case, $Agg_z[d_0]$ is a field of a sub-aggregate Agg_i. We may use the induction hypothesis to obtain an instantiation ν_i of Agg_i such that $Agg_z[d]$ belongs to ν_i. Let $e_i = \nu(y_i)$. Since $Agg_{y_i}[e_i]$ is not upwards isolated there exists a field $Agg_x[d]$ with a vertical pointer $Agg_x[d, y_i[l_i, v]]$ (for suitable l_i) with address field $Agg_{y_i}[e_i]$. Since $Agg_x[d]$ is not downwards isolated and since no pointer column of $Agg_x[d]$ is left or right isolated it follows easily that we may select for all $1 \leq j \neq i \leq h$ vertical pointers $Agg_x[d, y_j[l_j, v]]$ with address fields $Agg_{y_j}[e_j]$ that obey Condition 3 of Definition 6.14. By induction hypothesis, each of the fields $Agg_{y_j}[e_j]$ belongs to an instantiation ν_j of Agg_j $(1 \leq j \neq i \leq h)$. Combining the mappings ν_j for $1 \leq j \leq h$ and mapping x to d we obtain an instantiation ν of Agg such that $Agg_z[d_0]$ belongs to ν.

In the second case, where $Agg_z[d_0] = Agg_x[d]$ is in Agg_x we may directly use the fact that $Agg_x[d]$ is not downwards isolated and no pointer column of $Agg_x[d]$ is left or right isolated to conclude with the induction hypothesis that there exists an instantiation ν of Agg such that $Agg_z[d_0]$ belongs to ν.

The proof that each pointer column of Agg belongs to a suitable instantiation of Agg is analogous. $\square$

Summing up, we have seen that the output aggregate of the algorithm satisfies Condition 2 of Lemma 6.15. We now show *completeness* of the algorithm.

Lemma 6.21 *Each answer to Q may be obtained as an instantiation of the aggregate Agg that represents the output of Phase 2.*

Proof. Let $\nu : fr(Q) \to D$ be an answer to Q. If $\pi_Q = \langle x_k, \dots, x_0 \rangle$ is an inverted query path, then let $\pi = \langle d_m, d_{m-1}, \dots, d_0 \rangle$ denote the unique inverted document path with first (bottom-most) element $\nu(x_k)$. By assumption, π is one of the paths obtained by the index access for π_Q (cf. Remark 6.6). The mapping ν determines a match $\nu_{\pi_Q} : \{x_k, \dots, x_0\} \to \{d_m, d_{m-1}, \dots, d_0\}$ that can be used by the algorithm for successful alignment of the two paths. This shows that after finishing Part 1 of Phase 1 each record Agg_{x_i} will have a field $Agg_{x_i}[\nu(x_i)]$ $(0 \leq i \leq k)$, with vertical pointer $Agg_{x_i}[\nu(x_i), x_{i+1}[l, v]]$ to $Agg_{x_{i+1}}[\nu(x_{i+1})]$ for $i < k$ and suitable l. The combination of the mappings ν_{π_Q} for distinct inverted query paths π_Q, i.e, the mapping ν, satisfies Conditions 1 and 2 of Definition 6.14. Since ν is an answer to Q, and by definition of horizontal pointer addresses, it follows also that ν satisfies Condition 3 of Definition 6.14. Hence it defines an instantiation of the aggregate Agg_1 that is reached after Phase 1.

Clearly none of the fields/pointer columns that belong to ν are isolated in Agg_1. We now show by induction that none of the fields/pointer columns that belong to ν becomes quasi-isolated when applying *ELIM-PC* and *ELIM-F*. This shows that ν is an instantiation of the aggregate Agg_2 obtained as output of Phase 2 and finishes the proof. By Lemma 6.18 it suffices to show that none of the fields/pointer columns that belong to ν are added to *Isol-F* and *Isol-PC* respectively. A simple inspection of the procedures *ELIM-PC* and *ELIM-F* shows that a field/pointer column that belongs to ν can only be added to *Isol-F* and *Isol-PC* during a process where we actually eliminate (or colour red) another field/pointer column belonging to ν. This would mean that the latter field/pointer column had been added to *Isol-F* and *Isol-PC* before, which contradicts the induction hypothesis. □

It remains to prove *soundness* of the algorithm.

Lemma 6.22 *Each instantiation of the aggregate Agg that represents the output of Phase 2 is an answer to Q.*

Proof. Let ν be an instantiation of Agg. Let y be a child of x in Q, and let $e := \nu(y)$ and $d := \nu(x)$. Remark 6.6 and the tests in *SELECT-ANCESTORS* ensure that (e, d) satisfies all unary and binary index formulae imposed on (y, x) in Q. Condition 3 of Definition 6.14 ensures that ν satisfies all ordering constraints of Q. Hence ν is an answer to Q. □

7 Conclusion

We introduced a logical language for a variant of the Tree Matching formalism ([Kil92]), adding some flexibility[9] to the original formalism. In this context we introduced the new concept of a complete answer aggregate. This notion offers a concise presentation of the set of all answers to a query. We also showed how to compute a complete answer aggregate with a time/space complexity which is

[9] cf. Section 4.4.

optimal modulo a logarithmic factor. The complexity bounds indicate that the algorithm is interesting for database applications. The algorithm uses two index structures that support practical efficiency.

7.1 Related work

We briefly survey some related work. To the best of our knowledge no other approach uses the notion of aggregated answers so far, therefore we will not elaborate on this point. Note, however, that the models that are mentioned below use some kind of homomorphic mapping to define the notion of an answers to a query. Hence aggregate techniques could be useful for these systems as well.

Tree Matching As we argued in Section 4.4, our approach can be considered as a modification and extension of the original formalism. It preservs its power and strength: simplicity of the user interface, declarative semantics, rich query formalism and structured answers.

Dolores Dolores ([FGR98]) is a multimedia IR system that can handle arbitrary document structures. It is based on probabilistic logic and thus incorporates the notion of ranking. Queries are formulas and answers are variable assignments. The outstanding strength of Dolores is the capability to express uncertain knowledge. A major drawback is its architecture that translates the complete structure of queries and documents to probabilistic Datalog and thus fails to exploit the special features of tree-structured entities in query evaluation. The worst-case complexity of query evaluation is not mentioned, but due to the fact that Examples 5.1 and 5.2 can be expressed in Dolores, the time complexity must be at least $\mathcal{O}(n^q)$, where n is the size of the database and q the size of the query.

Lore Lore [MWA$^+$98] is an IR system for graph-structured documents. We review it here as one example for the research going on in the field of semistructured data (see [Abi97, Bun97, Suc98a] for surveys). Lore's sophisticated query evaluation mechanism involves the use of multiple index structures. The query language is SQL-based, answers are mappings. With the same argumentation as for Dolores we can infer that query evaluation is at least of time complexity $\mathcal{O}(n^q)$, where n is the size of the database and q the size of the query.

7.2 Open research issues

Some of the most important questions for further research are the following

1. *What are the most interesting relevance models for structured document retrieval, and how can complete answer aggregates help to provide an appropriate ranking?* The first question is not restricted to our formalism. We think that complete answer aggregates—or their logical descriptions in terms of complete answer formulae—might support some good ranking models, as we briefly indicated in the introduction.

2. *Given a very large database, is it possible to compute partial answer aggregates that guarantee a "sufficient" view of the most significant answers?*

3. *How can aggregates be combined with query modification and relevance feedback techniques?*

4. *How can aggregates be used for active manipulation of the database?*

5. *Can complete answer aggregates be used for other formalisms as well? For which ones? How?*

As a matter of fact it is also necessary to implement the algorithm and to check its practical behavior. One step to improve the efficiency of the algorithm that has not been described here is the use of filtering techniques for reducing the number of paths that are obtained from index accesses [MS99].

References

[Abi97] S. Abiteboul. Querying semi-structured data. In *Proc. 6th Int. Conf. on DB Theory, ICDT'97*, 1997.

[Bun97] Peter Buneman. Semistructured data. In ACM, editor, *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12–14, 1997, Tucson, Arizona*, pages 117–121. ACM Press, 1997.

[Bur92] F. J. Burkowski. An algebra for hierarchically organized text-dominated databases. *Information Processing & Management*, 28(3):333–348, 1992.

[BYN96] R. Baeza-Yates and G. Navarro. Integrating contents and structure in text retrieval. *SIGMOD Record*, 25(1):67–79, 1996.

[FGR98] N. Fuhr, N. Gövert, and T. Röllecke. DOLORES: A system for logic-based retrieval of multimedia objects. In *Proc. ACM SIGIR '98*, 1998.

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide-web: A survey. *SIGMOD Record*, 27(3), 1998.

[Gol90] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[GT87] G. Gonnet and F. Tompa. Mind your grammar: a new approach to modelling text. In *Proc. VLDB'87*, pages 339–346, 1987.

[ISO86] ISO. *Information Processing - Text and Office Systems - Standard General MarkUp Language (SGML)*. ISO8879, 1986.

[Kil92] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, Dept. of Computer Science, University of Helsinki, 1992.

[KM93] P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proc. ACM SIGIR'93*, pages 214–222, 1993.

[Loe94] A. Loeffen. Text databases: A survey of text models and systems. *SIGMOD Record*, 23(1):97–106, March 1994.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(3), 1997.

[Meu98] H. Meuss. Indexed tree matching with complete answer representations. In *Proc. Fourth Workshop on Principles of Digital Document Processing (PODDP'98)*, 1998.

[MS99] Holger Meuss and Christian Strohmaier. Improving index structures for structured document retrieval. In *21st Annual Colloquium on IR Research (IRSG'99)*, 1999.

[MSM93] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: the Penn Treebank. *Computational Linguistics*, 1993.

[MWA$^+$98] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajamaran. Indexing semistructured data. Technical report, Stanford University, Computer Science Department, 1998.

[NBY97] G. Navarro and R. Baeza-Yates. Proximal Nodes: A model to query document databases by contents and structure. *ACM Transactions on Information Systems*, 15(4):400–435, 1997.

[OMM98] Jürgen Oesterle and Petra Maier-Meyer. The gnop (german noun phrase) treebank. In *First International Conference on Language Resources and Evaluation, Granada, Spain*, pages 699 – 703, 1998.

[Suc98a] Dan Suciu. An overview of semistructured data. *SIGACT News*, 1998.

[Suc98b] Dan Suciu. Semistructured data and XML. In *Proc. of Int. Conf. on Foundations of Data Organization*, 1998.

[W3C98a] W3C. QL'98 - the query languages workshop, December 1998. `http://www.w3.org/TandS/QL/QL98`.

[W3C98b] World Wide Web Consortium: Extensible Markup Language (XML) 1.0. W3C Recommendation, February 1998. `http://http://www.w3.org/TR/REC-xml`.

[W3C98c] World Wide Web Consortium: HTML 4.0 Specification. W3C Recommendation, April 1998. `http://www.w3.org/TR/REC-html40/`.

Appendix

We give the proofs that were omitted in previous sections. For convenience, the respective lemmata and theorems are repeated.

Lemma 4.17 Let $\mathcal{D}_Q = (X_S, X_T, X_U, \rightharpoonup_Q, \overset{+}{\rightharpoonup}_Q, Lab_Q, I_Q)$ be the query tree of the tree query Q, let $\mathcal{D} = (D_S, D_T, \rightharpoonup_D, Lab_D, I_D)$ be a relational document structure. A mapping $\nu : \mathit{fr}(Q) \to D$ is a pseudo-homomorphism from $\mathcal{D}_Q$ in $\mathcal{D}$ iff ν is an answer to Q in $\mathcal{D}$.

Proof. Let $Q = (\psi \wedge c, \vec{x})$. We first show the direction "left to right":
Let $\nu : \mathit{fr}(Q) \to D$ be a pseudo-homomorphism from $\mathcal{D}_Q$ in $\mathcal{D}$. We show for every atom φ in $\psi \wedge c$ that $\mathcal{D} \models_\nu \varphi$. All argumentations follow the same line of references, first to Definition 4.12, then to Definition 4.15, and finally to Definition 4.2:

$x \lhd y$**:** With Definition 4.12 it follows: $x \rightharpoonup_Q y$. With Definition 4.15 we have $\nu(x) \rightharpoonup_D \nu(y)$. Finally Definition 4.2 implies $\mathcal{D} \models_\nu x \lhd y$.
$x \lhd^+ y$**:** Analogously.
w *in* x**:** It follows $w \in Lab_Q(x)$ and $x \in X_T$. Then we know that $Lab_D(\nu(x))$ contains w as substring and $\nu(x) \in D_T$. Therefore $\mathcal{D} \models_\nu w$ *in* x.
$M(x)$**:** Then $Lab_Q(x) = M$ and $x \in X_S$. Therefore $Lab_D(\nu(x)) = M$ and $\nu(x) \in D_S$. This results in $\mathcal{D} \models_\nu M(x)$.
$r(x_1, \ldots, x_k)$**:** Then $\langle x_1, \ldots, x_k \rangle \in I_Q(r)$. It follows that $\langle \nu(x_1), \ldots, \nu(x_k) \rangle \in I_D(r)$, and therefore $\mathcal{D} \models_\nu r(x_1, \ldots, x_k)$.

For the inverse direction let ν be an answer to Q in $\mathcal{D}$, i.e. $\mathcal{D} \models_\nu \psi \wedge c$. We show that ν is pseudo-homomorphism by validating each case in Definition 4.15.
As in the other direction, every case in the analysis of Definition 4.15 follows the same line of arguments: First a reference to Definition 4.12, then to Definition 4.2:
$x \in X_S$**:** With Definition 4.12 it follows that $M(x) \in \psi$ or $(x \lhd^{(+)} y) \in \psi$. In both cases Definition 4.2 together with Definition 3.1 guarantee that $\nu(x)$ is a structural node.
$x \in X_T$**:** It follows that $(w$ *in* $x) \in \psi$. Therefore $\nu(x)$ is a text node.
$x \rightharpoonup_Q y$**:** Then $(x \lhd y) \in \psi$, and therefore $\nu(x) \rightharpoonup_D \nu(y)$.
$x \rightharpoonup_Q^+ y$**:** Analogously.
$x \in X_S, Lab_Q(x) = M$**:** We have $M(x) \in \psi$ and then $Lab_D(\nu(x)) = M$.
$x \in X_T, w \in Lab_Q(x)$**:** Then $(w$ *in* $x) \in \psi$, and therefore $Lab_D(\nu(x))$ contains w.
$\langle y_1, \ldots, y_k \rangle$**:** It follows that $r(y_1, \ldots, y_k) \in c$, and therefore $\langle \nu(y_1), \ldots, \nu(y_k) \rangle \in I_D(r)$. □

Lemma 5.8 *Let Q be a simple tree query, let $\Delta_\epsilon^{(x_0)}$ be a dependent Q-instantiation formula for x_0, let $d_0, \ldots, d_k$ be elements of D. Then the following conditions are equivalent:*

1. *$\Delta_\epsilon^{(x_0)}$ has a subformula $\Delta_{(d_0,\ldots,d_{k-1})}^{(x_0,\ldots,x_k)}$ where d_k is a target candidate for x_k,*
2. *$\Delta_\epsilon^{(x_0)}$ has a subformula of the form $\delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$,*
3. *Q has formulae $x_0 \lhd^{(+)} x_1, \ldots, x_{k-1} \lhd^{(+)} x_k$ and $\{\langle x_i, d_i \rangle \mid 0 \leq i \leq k\}$ is a partial instantiation of $\Delta_\epsilon^{(x_0)}$.*

Proof. The equivalence "1 ⇔ 2" follows immediately from the definition of these subformulae. To prove the implication "2 ⇒ 3", let $\delta_{(d_0,\ldots,d_k)}^{(x_0,\ldots,x_k)}$ be a subformula of $\Delta_\epsilon^{(x_0)}$. The definition of these formulae shows that x_{i+1} is a child of x_i, for

$i = 0, \ldots, k-1$. Hence Q has formulae $x_0 \lhd^{(+)} x_1, \ldots, x_{k-1} \lhd^{(+)} x_k$. A trivial induction on k shows that $\{\langle x_i, d_i\rangle \mid 0 \leq i \leq k\}$ is a partial instantiation of $\Delta_\epsilon^{(x_0)}$. The inverse implication "3 $\Rightarrow$ 2" follows by a trivial induction on k. □

Lemma 5.11 *Let Δ_Q be a complete answer formula for the simple tree query Q. Then two subformulae of Δ_Q of the form $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d_0,\ldots,d_{k-1},d_k)}$ and $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d'_0,\ldots,d'_{k-1},d_k)}$ are always identical modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Proof. The proof of Theorem 5.10 shows that it suffices to verify the following: if $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d_0,\ldots,d_{k-1},d_k)}$ has a subformula of the form $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$, then $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d'_0,\ldots,d'_{k-1},d_k)}$ has a subformula of the form $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$ and vice versa.

Let $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d_0,\ldots,d_{k-1},d_k)}$ have a subformula of the form $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$. By Lemma 5.8, Δ_Q has partial instantiations of the form $\{\langle x_i, d_i\rangle \mid 0 \leq i \leq k+r\}$ and $\{\langle x_i, d'_i\rangle \mid 0 \leq i \leq k\}$ where $d_k = d'_k$. By Lemma 5.7 there exist answers ν_1 (resp.ν_2) of Q that extend the former (latter) partial instantiation. By Lemma 4.11 there exists an answer ν_3 to Q that coincides with answer ν_1 on the set of reflexive descendants of x_k and with answer ν_2 on all other variables in *fr*(Q). Answer ν_3 extends the partial instantiation $\{\langle x_i, d'_i\rangle \mid 0 \leq i \leq k-1\} \cup \{\langle x_i, d_i\rangle \mid k \leq i \leq k+r\}$. Hence $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d'_0,\ldots,d'_{k-1},d_k)}$ has a subformula of the form $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$. By symmetry the lemma follows. □

Before we can prove Theorem 5.23 some preparation is needed.

Lemma 7.1 *Let Q be a local tree query, let $\Delta_\epsilon^{x_0}$ be a dependent Q-instantiation formula. Then every partial instantiation of $\Delta_\epsilon^{x_0}$ can be extended to a total instantiation of $\Delta_\epsilon^{x_0}$. The set of total instantiations of $\Delta_\epsilon^{x_0}$ is non-empty.*

Proof. Follows immediately from the fact that restrictor sets for variables x with $h_Q(x) > 0$ as well as arbitrary sets of target candidates are always non-empty. □

Lemma 7.2 *Let Q be a local tree query, let $\Delta_\epsilon^{(x_0)}$ be a dependent Q-instantiation formula for x_0, let $d_0, \ldots, d_k$ be elements of D. Then the following conditions are equivalent:*

1. *$\Delta_\epsilon^{(x_0)}$ has a subformula $\Delta^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_{k-1})}$ where d_k is a target candidate for x_k,*
2. *$\Delta_\epsilon^{(x_0)}$ has a subformula of the form $\delta^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_k)}$,*

3. Q has formulae $x_0 \lhd^{(+)} x_1, \ldots, x_{k-1} \lhd^{(+)} x_k$ and $\{\langle x_i, d_i\rangle \mid 0 \leq i \leq k\}$ is a subset of a partial instantiation of $\Delta_\epsilon^{(x_0)}$ that does not instantiate any child of x_k in Q.

Proof. The equivalence "1 ⇔ 2" is trivial. Implication "2 ⇒ 3" follows from the contribution obligation mentioned after Definition 5.20 using induction on k. The converse direction "3 ⇒ 2" is trivial. □

Theorem 5.23 *For each local tree query Q a complete answer formula Δ_Q is unique modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Proof. Let $\Delta_\epsilon^{(x_0)}$ and $\Lambda_\epsilon^{(x_0)}$ be complete answer formulae for Q. Assume that $\Delta_\epsilon^{(x_0)}$ has a subformula $\delta^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_k)}$. If x_k is a leaf of Q it follows from Lemmata 7.1 and 7.2 that $\Lambda_\epsilon^{(x_0)}$ has a corresponding subformula $\lambda^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_k)}$. Assume now that $y_1, \ldots, y_h$ (for $h \geq 1$) denotes the set of children of x_k in Q. Let $(e_1, \ldots, e_h)$ be an element of the restrictor set of $\delta^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_k)}$. Using Lemmata 7.2 and 7.1 we see that Q has an answer 1 that maps x_i to d_i for $i = 1, \ldots, k$ and y_j to e_j for $j = 1, \ldots, h$. But then $\Lambda_\epsilon^{(x_0)}$ must have a subformula $\lambda^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_k)}$ where $(e_1, \ldots, e_h)$ is an element of the restrictor set, since otherwise answer 1 could not be obtained as an instantiation of $\Lambda_\epsilon^{(x_0)}$. By symmetry it follows that $\Delta_\epsilon^{(x_0)}$ and $\Lambda_\epsilon^{(x_0)}$ have corresponding subformulae of the form $\delta^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_k)}$ respectively $\delta^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_k)}$ with identical restrictor sets. Starting at the subformulae with maximal k it is then simple to prove by "inverse" induction that corresponding formulae $\delta^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_k)}$ and $\lambda^{(x_0,\ldots,x_k)}_{(d_0,\ldots,d_k)}$ are equal modulo associativity and commutativity of "$\wedge$". It follows that $\Delta_\epsilon^{(x_0)}$ and $\Lambda_\epsilon^{(x_0)}$ are equal modulo associativity and commutativity of "$\wedge$" and "$\vee$". □

Lemma 5.24 *Let Δ_Q be a complete answer formula for the local tree query Q. Then two subformulae of Δ_Q of the form $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d_0,\ldots,d_{k-1},d_k)}$ and $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d'_0,\ldots,d'_{k-1},d_k)}$ are always identical modulo associativity and commutativity of "$\wedge$" and "$\vee$".*

Proof. It suffices to verify the following: if $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d_0,\ldots,d_{k-1},d_k)}$ has a subformula of the form $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$, then $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d'_0,\ldots,d'_{k-1},d_k)}$ has a subformula of the form $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$ and vice versa. Moreover, if $h_Q(x_{k+r}) > 0$, then the restrictor sets of both formulae are identical.

Let $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$ be a subformula of $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d_0,\ldots,d_{k-1},d_k)}$. By Lemma 5.8, Δ_Q has partial instantiations that extend the mappings $\{\langle x_i, d_i\rangle \mid 0 \leq i \leq k+r\}$ and $\{\langle x_i, d'_i\rangle \mid 0 \leq i \leq k\}$ where $d_k = d'_k$. By Lemma 5.7 there exists an answer 1 (resp. 2) of Q that extends the former (latter) partial instantiation. By Lemma 4.11

there exists an answer 3 of Q that coincides with answer 1 on the set of reflexive descendants of x_k and with answer 2 on all other variables in *fr*(Q). Answer 3 extends the mapping $\{\langle x_i, d'_i\rangle \mid 0 \leq i \leq k-1\} \cup \{\langle x_i, d_i\rangle \mid k \leq i \leq k+r\}$. But then $\delta^{(x_0,\ldots,x_{k-1},x_k)}_{(d'_0,\ldots,d'_{k-1},d_k)}$ must have a subformula of the form $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$. By symmetry the first condition mentioned above follows.

Assume that that both subformulae exist and that $h_Q(x_{k+r}) > 0$. Let $y_1,\ldots,y_h$ be the sequence of children of x_{k+r} and let $(e_1,\ldots,e_h)$ be an element of the restrictor set of $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d_0,\ldots,d_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$. It follows from Lemma 5.8 and Lemma 5.7 that Q has an answer 4 that extends the mapping $\{\langle x_i, d_i\rangle \mid 0 \leq i \leq k+r\} \cup \{\langle y_i, e_i\rangle \mid i = 1,\ldots,h\}$. Lemma 4.11 shows that there exists an answer 5 of Q that extends the mapping $\{\langle x_i, d'_i\rangle \mid 0 \leq i \leq k-1\} \cup \{\langle x_i, d_i\rangle \mid k \leq i \leq k+r\} \cup \{\langle y_i, e_i\rangle \mid i = 1,\ldots,h\}$. But then $(e_1,\ldots,e_h)$ must be an element of the restrictor set of $\delta^{(x_0,\ldots,x_{k-1},x_k,x_{k+1},\ldots,x_{k+r})}_{(d'_0,\ldots,d'_{k-1},d_k,d_{k+1},\ldots,d_{k+r})}$. By symmetry it follows that both subformulae mentioned above have the same restrictor set. $\square$

Lemma 5.26 *Let $Q = (\psi \wedge c, \vec{x})$. In the situation of Definition 5.25, let R denote the restrictor set of $\delta_x(d)$, for $i = 1,\ldots,h$ let D_i be the set of target candidates for y_i in $\Delta_{x,y_i}(d)$. Then R is the set of all tuples $(e_1,\ldots,e_h) \in D_1 \times \cdots \times D_h$ where $(e_1,\ldots,e_h)$ satisfies all non Q-simple constraints $r(x, y_{i_1},\ldots,y_{i_r})$ of c relative to d where $\{y_{i_1},\ldots,y_{i_r}\} \subseteq \{y_1,\ldots,y_h\}$.*

Proof. Let $(e_1,\ldots,e_h) \in R$. By definition, $(e_1,\ldots,e_h) \in D_1 \times \cdots \times D_h$. It follows from Lemma 7.1 and Lemma 7.2 that Q has an answer that extends the mapping $\{\langle x_i, d_i\rangle \mid i = 1,\ldots,k\} \cup \{\langle y_i, e_i\rangle \mid i = 1,\ldots,h\}$. This shows that $(d_k, e_1,\ldots,e_h)$ satisfies all constraints $r(x_k, y_{i_1},\ldots,y_{i_r})$ in c where $\{y_{i_1},\ldots,y_{i_r}\} \subseteq \{y_1,\ldots,y_h\}$.

Conversely let $(e_1,\ldots,e_h) \in D_1 \times \cdots \times D_h$, assume that $(d_k, e_1,\ldots,e_h)$ satisfies all non Q-simple constraints $r(x_k, y_{i_1},\ldots,y_{i_r})$ in c where $\{y_{i_1},\ldots,y_{i_r}\} \subseteq \{y_1,\ldots,y_h\}$. Assume, to get a contradiction, that $(e_1,\ldots,e_h) \notin R$. Replacing R with $R \cup \{(e_1,\ldots,e_h)\}$ we would get a dependent Q-instantiation formula with a larger set of instantiations where still each instantiation is an answer to Q. In fact, since we did not modify any set of target candidates the new Q-instantiation formula leads to instantiations that satisfy all Q-simple constraints and $\mathcal{L}$-formulae of the query. This would mean that Δ_Q is not a complete answer formula. $\square$