Semantic Querying of Tree-Structured Data Sources Using Partially Specified Tree Patterns

Dimitri Theodoratos
Dept. of CS
NJIT
dth@cs.njit.edu

Theodore Dalamagas
School of EE and CE
NTUA
dalamag@dblab.ntua.gr

Antonis Koufopoulos
School of EE and CE
NTUA
akoufop@dblab.ntua.gr

Narain Gehani
Dept. of CS
NJIT
gehani@njit.edu

ABSTRACT

Nowadays, huge volumes of data are organized or exported in a tree-structured form. Querying capabilities are provided through queries that are based on branching path expression. Even for a single knowledge domain structural differences raise difficulties for querying data sources in a uniform way. In this paper, we present a method for semantically querying tree-structured data sources using partially specified tree patterns. Based on dimensions which are sets of semantically related nodes in tree structures, we define dimension graphs. Dimension graphs can be automatically extracted from trees and abstract their structural information. They are semantically rich constructs that support the formulation of queries and their efficient evaluation. We design a tree-pattern query language to query multiple tree-structured data sources. A central feature of this language is that the structure can be specified fully, partially, or not at all in the queries. Therefore, it can be used to query multiple trees with structural differences. We study the derivation of structural expressions in queries by introducing a set of inference rules for structural expressions. We define two types of query unsatisfiability and we provide necessary and sufficient conditions for checking each of them. Our approach is validated through experimental evaluation.

Categories and Subject Descriptors: H.2.3 Languages, H.2.4 Information Search and Retrieval, H.3.3 Systems

General Terms: management, experimentation, languages.

Keywords: tree-structured data, tree-pattern queries, XML, query evaluation, query satisfiability

1. INTRODUCTION

Nowadays, huge volumes of data are posted and retrieved through the Web. Tree structures provide the most popular means for organizing the information. Taxonomies of thematic categories, concept hierarchies, e-commerce product catalogs are examples of such structures. The Extensible Markup Language (XML) [1] is by far the most prominent example. Even if XML data is stored in relational databases, export mechanisms make it publicly available in tree-structure format [7].

Queries on tree-structured data are mainly based on the usage of branching path expressions. XPath [2] is a language that uses path expressions to navigate through the tree structure of an XML document, binding nodes to variables. It lies at the core of W3C language proposals for XML querying and transformation (e.g. XQuery [3]).

The problem. The major challenge in this context is to provide the means for the effective and efficient querying of multiple tree-structured data sources. This goal is hampered by the semistructured nature of this data. When there is no schema or when the structure of the data is not fully available, it is very difficult or even impossible for the user to specify the right query on a single tree structure. Querying multiple tree-structured data sources usually requires resolving structural differences. This is due to the different possible ways of organizing the same data in trees. For instance, a structural difference exists when a category *Painter* appears in a museum catalog but does not appear in another. Another structural difference appears when, for example, a museum exports an XML document that records `Picasso`'s `self-portrait exhibition` along the element sequence $Artist$, $Work\_of\_Art$, $Exposure$ but records other artist's creations along the element sequence $Exposure$, $Artist$, $Work\_of\_Art$. Such differences raise major difficulties for querying multiple tree-structured data sources.

Relevant approaches. One can distinguish three approaches to the problem that are based on branching path expressions. A traditional information integration approach attempts to cope with this problem by providing a global structure. Mapping rules are defined between the global structure and the local structures used in the sources [10]. For example, given the rules *(Exposure/Work_of_Art)* $\rightarrow$ *(Work_of_Art/Exposure)* for one source, query *Artist/Exposure/Work_of_Art* will become *Artist/Work_of_Art/ Exposure* in order to match another tree structure. Such approaches require extensive manual effort since the global schema is difficult to construct and the rules should be hard-coded in the integration application. Moreover, the user should have exact knowledge of the global structure.

A naive approach executes the queries that result from all possible orderings of query elements and combine the answers. Clearly such an approach is unfeasible even for medium size queries.

CIKM'05, October 31–November 5, 2005, Bremen, Germany.

An approximate approach evokes tree pattern query relaxation methods. For instance, in [5], when the user explicitly specifies a sequence for elements *Exposure*, *Work_of_Art* and *Artist* to retrieve data, the system can relax the initial query, by removing the element *Work_of_Art* and by replacing it by a descendant relationship between *Exposure* and *Artist*. Nevertheless, the answer is not exact with respect to the query posed by the user.

Approaches that are not based on branching path expressions but on keyword search [13, 11, 16] avoid the problem of unknown structure or the problem of multiple differently structured data sources. In this case knowledge of the structure is not required for the queries. However, the total absence of structure has a number of important drawbacks, since the user can neither impose structural conditions to filter out answers nor specify structural constraints within the query to speed up the computation of the answer.

Our approach. In this paper we present a different approach by adopting a tree-structured data model that exploits semantic information, and by introducing a tree-pattern query language that allows a partial specification of the tree-patterns. Tree-structured data are called here value trees. We introduce dimensions which are sets of semantically related nodes in value trees. Based on dimensions, we define dimension graphs. Dimension graphs are not schemas but semantically rich constructs that abstract the structural information of the value trees. Dimension graphs can be automatically extracted from value trees and support the formulation of the queries and their evaluation. Queries are specified on dimensions and are not cast on the structure of a specific value tree. Therefore, the query language can be used to effectively query tree-structured data whose structure is not fully known or to integrate tree-structured data that have structural differences. Our approach differs from traditional data integration techniques in that it does not require the manual definition of transformation rules between a virtual global tree structure and the structures in the local data sources. It differs from query relaxation techniques, since it returns exact and not approximate answers. Finally, it differs from keyword search approaches since it allows partial or complete specification of the tree structure in the queries.

Contribution. The main contributions of this paper are the following:

- We design a query language to query value trees. Queries in this language provide a partial specification of a tree of dimensions annotated with sets of values. The user has the option to specify structural constraints - fully, partially, or not at all - when issuing queries on dimensions.
- Besides declarative, we also provide procedural semantics to our query language. Query evaluation is based on branching path expressions to be evaluated on the value trees. These expressions are generated using a dimension graph by determining completely specified tree-pattern queries (called dimension trees) that can possibly generate non-empty answers.
- The partial specification of the structure of the queries allows structural expressions to be derived from those explicitly specified in a query. We study the inference of structural expressions in queries by providing inference rules used for detecting unsatisfiable queries and efficiently computing dimension trees.
- We define two types of query unsatisfiability to identify queries that always produce empty answers. Unsatisfiable queries can be detected during the generation of dimension trees. This saves the bulk of the workload of query evaluation since dimension graphs are typically much smaller than their value trees.
- We have implemented a prototype system to evaluate our approach. Our system is built on top of an XQuery engine. We carried out several experiments to compare our approach to one that does not exploit dimension graphs in the evaluation of queries on value trees. Our results demonstrate the superiority of our approach.

2. RELATED WORK

Many systems support querying and integration of structured data using a predefined global structure and defining mapping rules between this structure and the local structures used in the sources [14]. In the area of tree-structured data, the Xyleme system [10] exploits XML views to cope with the problem of integrating XML data sources. The Agora system [17] translates XQuery expressions over a given global XML schema to SQL queries on local data sources. In [9], YAT queries are posed on a global schema, processed using mapping rules, and then evaluated in the data sources. Our approach can support querying of multiple tree-structured data sources through the formulation of queries on dimensions spanning all the local value trees. Therefore, it does not require the manual definition of mapping rules.

Schema-based descriptions have been suggested for semistructured databases [4]. Dataguides [15] are structural summaries for semistructured data, storing statistics about paths and nodes, and enabling query optimization. Statistical synopses for graph-structured XML databases are suggested in [18]. In [6], graph schemas are introduced to formulate and optimize queries for semistructured data. These approaches are purely syntactic. In contrast to our approach, they do not exploit semantic information. Query formulation is strictly dependent on the knowledge of structural irregularities in tree-structured data. In our approach, queries are not restricted by the structure of data. A problem similar to the one addressed in this paper is studied in [16]. The suggested language does not directly allow the specification of several nodes on a tree pattern path without specifying an order for these nodes. In [19, 12], a query language to query value trees without fully specifying the pattern is defined. However, the adopted data model is more appropriate for classification hierarchies while the query language supports only path patterns and not tree patterns.

3. DATA MODEL

We present below our data model for tree-structured data.

3.1 Dimensions and Value Trees

We assume an infinite set of values V that includes a special value r. For instance, "P. Picasso", "MoMA", and "self-portrait" are values of an application involving artists, museums, and works of art. A *dimension set* over V is a partition $\mathcal{D}$ of V that includes a set whose single element is value r. Each element of $\mathcal{D}$ is called *dimension* of $\mathcal{D}$. The dimensions in $\mathcal{D}$ are assigned distinct names. In particular, the dimension $\{r\}$ is named R. Intuitively, a dimension is

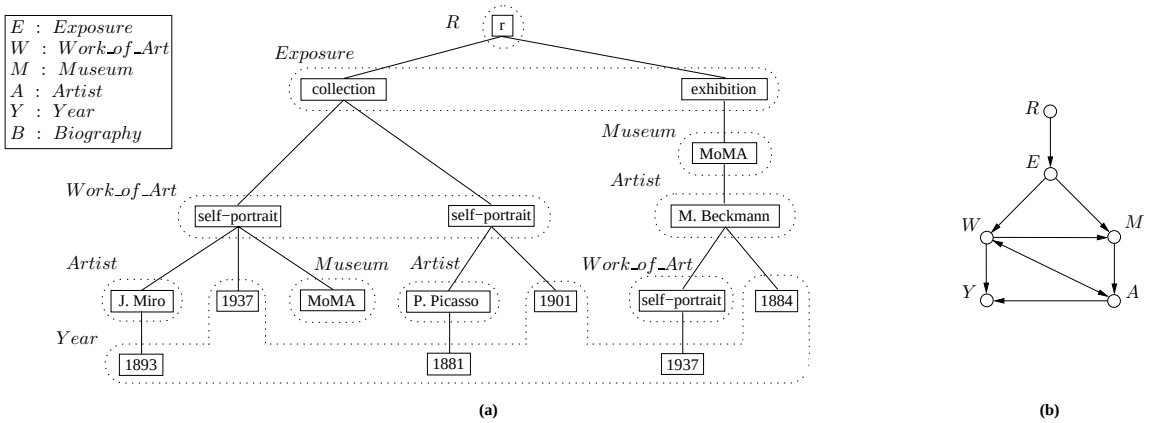

Figure 1: (a) Value tree T_1, (b) Dimension graph $\mathcal{G}_1$ of T_1.

a set of semantically related values. For instance, *Artist* can be a dimension that includes values "P. Picasso", "J. Miro", and "M. Beckmann". For the needs of this paper, we assume a single fixed dimension set $\mathcal{D}$.

DEFINITION 3.1. *A* value tree *over $\mathcal{D}$ is a rooted node-labeled tree T, such that: (a) Each node label in T belongs to V, (b) Value r labels only the root of T, and (c) There are no two nodes on a path in T labeled by values that belong to the same dimension in $\mathcal{D}$.* □

EXAMPLE 3.1. *Figure 1(a) shows a value tree T_1 that conveys information about museums, artists, and works of art. Closed dotted lines labeled by dimensions are used to show the partitioning of values in T_1 into dimensions. The same dimension might label multiple closed dotted lines in T_1. From a structural point of view, the information in T_1 is organized differently under nodes* collection *and* exhibition *with respect to the dimensions.* □

The semantic interpretation of the values of a value tree into dimensions can be provided by a user, or by an ontology. Different applications may require and apply different partitions of the values of the same value tree.

3.2 Dimension Graphs

The values of some dimension may not be children or descendants of any value of some other dimension in a value tree. For instance, no value of dimension *Exposure* in the value tree T_1 of Figure 1 is a child or descendant of a value of any of the other dimensions in T_1 except R. We use the concept of dimension graph to capture this type of relationship between dimensions in a value tree.

DEFINITION 3.2. *Let T be a value tree over $\mathcal{D}$. A* dimension graph *of T is a graph (N, E), where N is a set of nodes and E is a set of edges defined as follows: (a) There is a node D in N if and only if there is a value in T that belongs to dimension D, and (b) There is a directed edge in E from node D_i to node D_j if and only if there are nodes n_i and n_j in T labeled by values $v_i \in D_i$ and $v_j \in D_j$, respectively, such that n_j is a child node of n_i in T. If $\mathcal{G}$ is a dimension graph of a value tree T, we say that T* underlies *$\mathcal{G}$.* □

Figure 1(b) shows the dimension graph of the value tree T_1 of Figure 1(a). Dimension names are abbreviated as shown in the same figure. The dimension graph of a value tree may have cycles. A double headed edge between two dimensions (e.g. the edge between W and A in Figure 1(b)) denotes a trivial cycle.

The following proposition provides properties that fully characterize dimension graphs on value trees.

PROPOSITION 3.1. *Let $\mathcal{D}$ be a dimension set. A directed graph $\mathcal{G}$ whose nodes are dimensions is a dimension graph of a value tree over $\mathcal{D}$ if and only if the following properties hold: (a) Graph $\mathcal{G}$ does not have disconnected components. (b) There is exactly one node in $\mathcal{G}$ having only outgoing edges. We call this node* root *of $\mathcal{G}$. (c) For every directed edge in $\mathcal{G}$ there is a simple path from the root of $\mathcal{G}$ that comprises this edge.* □

4. QUERY LANGUAGE

Queries are issued on dimension sets and are evaluated on values trees. We require that the evaluation of a query on a value tree yields a value tree. One reason for this closed-form requirement on queries is that it allows query composition.

4.1 Syntax

A query on a dimension set provides a (possibly partial) specification of a tree of dimensions annotated with sets of values. The tree is rooted at dimension R. A query specifies such a tree through a set of (possibly partially specified) paths from the root of the tree. For brievity's sake, in the following, *PSP* stands for *partially specified path.* Each PSP is defined in a query by a set of annotated dimensions, and a set of precedence relationships (child and descendant relationships) among these annotated dimensions. A query further indicates nodes (annotated dimensions) that are shared among different PSPs in the query tree. It also identifies a distinguished PSP called output path. The formal definition follows.

DEFINITION 4.1. *A* query *on a dimension set $\mathcal{D}$ is a triple $(\mathcal{P}, \mathcal{S}, o)$, where:*

(a) $\mathcal{P}$ is a nonempty set of triples $(p, \mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ and $\mathcal{R}$ define a PSP as explained below, and p is a distinct name for this PSP. Since PSP names are distinct, we identify PSPs with their names.

(a1) $\mathcal{A}$ is a set of expressions of the from $D[p] = V$, where D is a dimension in $\mathcal{D}$, and V is a set of values of dimension D or a question mark ("?"). These expressions

are called annotating expressions *of* p. *If the expression* $D[p] = V$ *belongs to* $\mathcal{A}$ *we say that* D *is* annotated *in* p *and* V *is its* annotation. *A dimension can be annotated only once in a PSP* p. *Without mentioning it explicitly, we assume that dimension* R *is annotated with a '?' in every PSP. Set* $\mathcal{A}$ *can be empty.*

(a2) $\mathcal{R}$ *is a set of expressions of the form* $D_i[p] \rightarrow D_j[p]$ *or* $D_i[p] \Rightarrow D_j[p]$, *where* D_i *is an annotated dimension in* $\mathcal{A}$ *or* R, *and* D_j *is an annotated dimension in* $\mathcal{A}$. *These expressions are called* precedence relationships *of* p. *Set* $\mathcal{R}$ *can be empty.*

(b) $\mathcal{S}$ *is a set of expressions of the form* $D[p_i] = D[p_j]$, *where* p_i *and* p_j *are PSPs in* $\mathcal{P}$, *and* D *is a dimension annotated in* p_i *and* p_j. *These expressions are called* node sharing expressions *of* Q. *Set* $\mathcal{S}$ *can be empty.*

(c) o *is the name of one of the PSPs in* $\mathcal{P}$. *This PSP is called* output PSP. □

EXAMPLE 4.1. *Let the dimension set* $\mathcal{D}$ *comprise the dimensions shown in Figure 1. Let* $Q_1 = (\mathcal{P}, \mathcal{S}, p_2)$ *be a query on* $\mathcal{D}$, *where* $\mathcal{P} = \{(p_1, \mathcal{A}_1, \mathcal{R}_1), (p_2, \mathcal{A}_2, \mathcal{R}_2)\}$, $\mathcal{A}_1 = \{W[p_1] = ?, Y[p_1] = \{1937, 1884\}\}$, $\mathcal{R}_1 = \{W[p_1] \Rightarrow Y[p_1]\}$, $\mathcal{A}_2 = \{W[p_2] = \{\textit{self-portrait}\}, E[p_2] = ?, A = ?\}$, $\mathcal{R}_2 = \emptyset$, *and* $\mathcal{S} = \{W[p_1] = W[p_2]\}$. □

Query Q_1 of Example 4.1 is graphically represented in Figure 2(a).

In a graphical representation of a query Q, each PSP of Q is represented as a (not necessarily connected) graph. The name of each PSP is shown below the corresponding PSP graph. In particular, the name of the output PSP of Q is preceded by a $\star$. Each annotated dimension in a PSP is represented by a node labeled by the corresponding annotating expression. PSP names are omitted in the annotating expressions for succinctness. Child and descendant precedence relationships in a PSP are depicted using single ($\rightarrow$) and double ($\Rightarrow$) arrows between the respective nodes in the PSP graph. Two nodes (annotated dimensions) in different PSP graphs that participate in a node sharing expression of Q are linked in its graphical representation with a straight line labeled by the symbol '$\equiv$'.

Figures 2(b) and (c) show two other queries, query Q_2 and query Q_3, both of them similar to query Q_1.

4.2 Semantics

The answer of a query Q on a value tree T is a value tree, subtree of T. Every path from the root to a leaf of T is the image of the output path of Q under an embedding of Q into T that preserves the precedence relationships and nodes sharing expressions of Q. More formally:

DEFINITION 4.2. *Let* T *be a value tree over a dimension set* $\mathcal{D}$, *and* Q *be a query on* $\mathcal{D}$. *An* embedding *of* Q *into* T *is a mapping* M *of the annotated dimensions of the PSPs of* Q *to nodes in* T *such that:*

(a) The annotated dimensions of a PSP in Q *are mapped to nodes in* T *that are on the same path from the root of* T.

(b) For every annotating expression $D[p_i] = V$ *in* Q, *the label of* $M(D[p_i])$ *is a value in* V, *if* V *is a set, and it is a value of* D, *if* V *is a "?".*

(c) For every precedence relationship $D_j[p_i] \rightarrow D_k[p_i]$ *(resp.* $D_j[p_i] \Rightarrow D_k[p_i]$*) in* Q, $M(D_k[p_i])$ *is a child (resp. descendant) of* $M(D_j[p_i])$ *in* T.

(d) For every node sharing expression $D[p_i] \equiv D[p_j]$ *in* Q, $M(D[p_i])$ *and* $M(D[p_j])$ *coincide.* □

Given an embedding M of a query Q into a value tree T, and a PSP p in Q, the path from the root of T that comprises all the images of the annotated dimensions of p under M and ends in one of them is called *image* of p under M and is denoted $M(p)$. Notice that more than one PSP of Q may have their image in the same root-to-leaf path of T.

DEFINITION 4.3. *Let* T *be a value tree over a dimension set* $\mathcal{D}$, *and* $Q = (\mathcal{P}, \mathcal{S}, o)$ *be a query on* $\mathcal{D}$. *The* answer *of* Q *on* T *is a subtree* T' *of* T *such that:*

(a) For every embedding of Q *into* T, *the image of the output PSP of* Q *is in* T'.

(b) Every root-to-leaf path of T' *is the image of the output PSP of* Q *under an embedding of* Q *into* T.

If there is no such a subtree T', *we say that the answer of* Q *on* T *is empty.* □

Note that annotating a dimension with a "?" in a PSP of a query is different than not annotating this node at all.

EXAMPLE 4.2. *Consider the queries* Q_1, Q_2 *and* Q_3 *of Figures 2(a), (b), and (c) respectively, and the value tree* T_1 *of Figure 1(a). The answer of* Q_1 *on* T_1 *is shown in Figure 3(a). There are two embeddings of* Q_1 *into* T_1 *which result in two distinct root-to-leaf paths in the answer of* Q_1 *on* T_1. *Notice that one of these embeddings maps the dimensions of both PSPs of* Q_1 *to nodes on the same path from the root of* T_1 (r/exhibition/MoMa/M. Beckmann/self-portrait/1937).

Figure 3(b) shows the answer of Q_2 *on* T_1. *Query* Q_2 *is more restricted than* Q_1. *Only one of the embeddings of* Q_1 *into* T_1 *is also an embedding of* Q_2 *into* T_1. *The answer of* Q_2 *on* T_1 *has a single root-to-leaf path.*

Query Q_3 *is also more restricted than* Q_1. *It does not have any embedding into* T_1. *Its answer on* T_1 *is empty.* □

5. QUERY EVALUATION

The goal of this section is twofold: first to provide procedural semantics to the query language and second to show that query evaluation can be easily implemented on top of an XQuery [3] engine by generating branching path expressions. Such an implementation can benefit from the optimization techniques developed so far for XQuery. The evaluation of queries is based on the concept of dimension tree of a query on a dimension graph.

5.1 Dimension trees

Given a query Q and a dimension graph $\mathcal{G}$, a dimension tree of Q on $\mathcal{G}$ is a tree U whose nodes are labeled by annotated dimensions and its root is labeled by the annotated dimension R. It also has a distinguished node called *output node* that defines a path from its root called *output path*. A dimension tree U of Q on $\mathcal{G}$ satisfies two properties. First, there is a mapping m from the nodes of U to the nodes of $\mathcal{G}$ that is root-preserving, and respects labeling dimensions and child relationships. Second, there is a mapping m' from the annotated nodes of Q to the nodes of U that respects labeling dimensions, annotations (existentially), precedence relationships, and node sharing expressions. Intuitively, a dimension tree for Q on $\mathcal{G}$ represents an embedding of Q into $\mathcal{G}$ that respects labeling dimensions, precedence relationships, node sharing expressions and annotations (existentially). This embedding is the composition of m' and m. The dimensions trees of Q on $\mathcal{G}$ represent all such possible embeddings of Q into $\mathcal{G}$. We skip the formal definition because of lack of space.

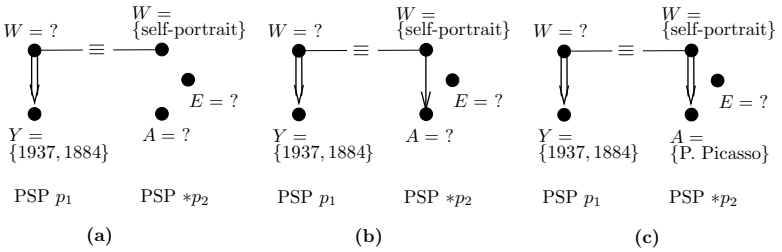

Figure 2: (a) Query Q_1, (b) Query Q_2, (c) Query Q_3

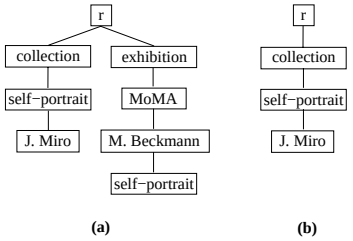

Figure 3: Answer of (a) Q_1, (b) Q_2

EXAMPLE 5.1. *Consider query Q_1 of Figure 2(a) and the dimension graph $\mathcal{G}_1$ of Figure 1(b). Figure 4 shows the dimension trees of Q_1 on $\mathcal{G}_1$. The output nodes of the dimension trees are shown filled with gray color. All the nodes of a dimension tree are annotated. Annotations that are '?' are not shown in the figure. Notice that mapping m maps the nodes of the two roof-to-leaf paths in U_7 to the nodes of the image in $\mathcal{G}$, of the longest of these two paths. Of the seven dimension trees of Figure 4, the first three (U_1, U_2, and U_3) are the dimension trees of query Q_2 of Figure 2(b), while the first six (U_1 to U_6) would be the dimension trees of query Q_3 of Figure 2(c) if their output node labeled by dimension A were annotated with the set* {P. Picasso}. □

5.2 Evaluating queries using dimension trees

A dimension tree U of a query Q on a dimension graph $\mathcal{G}$ can also be viewed as a (structurally) completely specified query. Complete specification means that the query forms a tree pattern (without missing edges) involving only parent-child (and not ancestor-descendant) relationships. Thus, given dimension graph $\mathcal{G}$, query Q represents a set of completely specified queries. We show in this subsection that the answer of Q on a value tree T underlying $\mathcal{G}$ can be constructed using the answers on T of all these completely specified queries.

Given a dimension tree U, the corresponding (completely specified) query Q_U can be formed in a straightforward way by letting the root-to-leaf paths of U define the PSPs of query Q_U. If the output node of U is a leaf node, the output PSP of Q_U is the PSP in Q_U that corresponds to the output root-to-leaf path of U. Otherwise, the output PSP of Q_U is an additional PSP defined by the output path of U. The labeling dimensions and annotations of the nodes in the root-to-leaf paths of U define dimensions and their annotations in Q_U. The edges in the root-to-leaf paths of U induce the child precedence relationships of Q_U. Finally, the common nodes of distinct root-to-leaf paths in U induce the node sharing expressions of Q_U. In the following, we might identify a dimension tree with its corresponding query.

We assume that nodes in a value tree have a unique node identifier and these node identifiers are preserved in the answers of a query on this value tree. Let T_1 and T_2 be two value trees, subtrees of the same value tree. The *merge* of T_1 and T_2, denoted $T_1 \oplus T_2$, is the value tree obtained by merging the nodes of T_1 and T_2 that have the same node identifiers (and the edges between merged nodes). Clearly, this operation is associative and commutative. If $T_1, \ldots T_k, k > 0$, are value trees, subtrees of the same value tree, $\oplus_{i\in[1,k]} T_i$ denotes the merge of $T_1, \ldots T_k$.

PROPOSITION 5.1. *Let Q be a query on a dimension set $\mathcal{D}$, T be a value tree on $\mathcal{D}$ and $\mathcal{G}$ be the dimension graph of T. Let $U_1, \ldots, U_n, n \geq 1$, be the dimension trees of Q on $\mathcal{G}$. If $T_1, \ldots, T_n$ are the answers of $U_1, \ldots, U_n$ on T, respectively, then the answer of Q on T is $\oplus_{i\in[1,k]} T_i$.* □

EXAMPLE 5.2. *Consider the dimension trees $U_1, \ldots, U_7$ of Figure 4. These are the dimension trees of query Q_1 of Figure 2(a) on the dimension graph $\mathcal{G}_1$ of Figure 1(b). The answer of U_1 on the value tree T_1 of Figure 1(a) is the left root-to-leaf path, while that of of U_7 is the right root-to-leaf path in the value tree of Figure 3(a). Dimension trees $U_2, \ldots, U_6$ on T_1 return empty answers. Therefore, the answer of Q_2 on T_1 is the merge of the answers of U_1 and U_7 on T_1. This merge is shown in Figure 3(a).* □

5.3 Branching path expression generation

Since dimension trees are completely specified queries, their answers can be computed by branching path expressions. We assume that the results of these branching path expressions are not nodes but paths from the root of the value tree.

The fragment of branching path expressions that we consider here involves node tests, the child axis (/), branching path predicates (specified between square brackets '[' and ']'), and the union (i.e. logical or) symbol (|). The expression $(v_1|\ldots|v_k)$ specifies any node label in the set $\{v_1|\ldots|v_k\}$. Our fragment involves also a restricted type of wildcard ($\star$): $\star_D$ denotes the expression $(v_1|\ldots|v_n)$, where $\{v_1, \ldots, v_n\} = D$. The predicates may also involve the Boolean connective `and`.

We assume that membership of values to dimensions is implemented by maintaining a mapping from dimensions to sets of values. This implementation supports the efficient evaluation of expressions of the form $\star_D$ at a node n of a value tree. Clearly, $\star_D$ is satisfied at n when n is labeled by any value of dimension D. Thus, we assume that the dimension of the label of a node of a value tree is stored with the label or is directly available at the node.

Let U be a dimension tree U. Every node n of U is mapped to a construct $t(n)$ (simple node test or union of node tests or restricted wildcard) using a function t on the nodes of U t defined below:

$$t(n) = \begin{cases} (\mathtt{v_1}|\ldots|\mathtt{v_k}) & \text{if } n \text{ is annotated in } U \text{ with the set } \{v_1, \ldots, v_k\},\ k \geq 1. \\ \star_{\mathtt{D}} & \text{if } n \text{ is annotated by a ? and is labeled by dimension } D \text{ in } U. \end{cases}$$

A branching path expression that computes the answer of U can now be easily constructed through a depth first traversal of U that uses function t defined above.

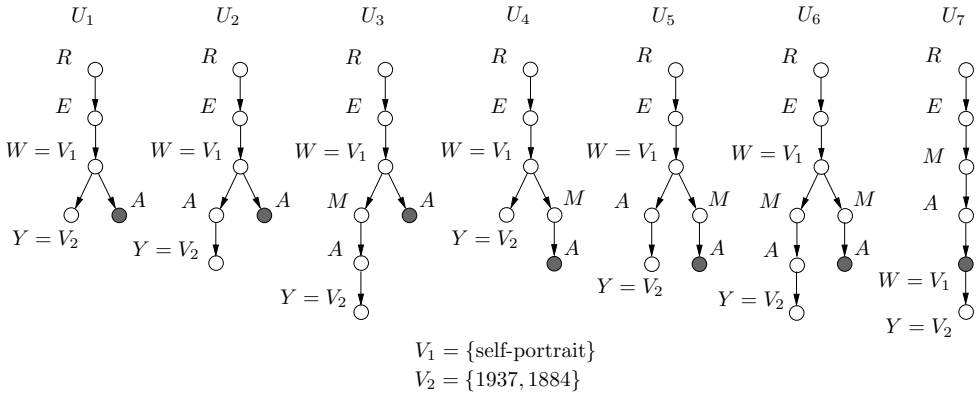

Figure 4: The dimension trees of query Q_1 on dimension graph $\mathcal{G}_1$

EXAMPLE 5.3. *The following branching path expressions correspond to the dimension trees U_3 and U_7 respectively:*
e_3 : r/ $\star_{\text{E}}$ /self-portrait[$\star_{\text{M}}$/ $\star_{\text{A}}$ /(1937|1884)]/$\star_{\text{A}}$
e_7 : r/ $\star_{\text{E}}$ / $\star_{\text{M}}$ / $\star_{\text{A}}$ /self-portrait[(1937|1884)]. □

6. QUERY UNSATISFIABILITY

Detecting unsatisfiable queries stops their evaluation at an early stage or even before it starts, thus saving valuable query computation time. When a dimension graph is given, an otherwise satisfiable query may be unsatisfiable with respect to all the value trees that underlie the dimension graph. This type of query unsatisfiability is expected to occur frequently in the environment of our approach because: (a) the user might be unaware of the structure of a data source, and (b) different data sources may have substantial structural differences and irregularities and a query which is issued against all of them may very well be unsatisfiable with respect to the dimension graphs of some of them. Detecting this type of query unsatisfiability using a dimension graph saves again valuable query computation time since it prevents accessing the value trees which are, in general, much larger than their dimension graphs.

In the context of our approach, checking query unsatisfiability becomes more complex. The reason is that the partial specification of the structure in the queries allows non-trivial structural expressions to be *inferred* from those explicitly specified in the queries. These inferred structural expressions need to be taken into account in determining query unsatisfiability. In this section, we first discuss structural expression inference before focusing on query unsatisfiability.

6.1 Structural expression inference

Given a query, new precedence relationships and/or node sharing expressions (collectively referred to as *structural expressions*) may be derived.

DEFINITION 6.1. *Let $\mathcal{E}$ be a set of structural expressions of a query Q on a dimension set $\mathcal{D}$, and e be a structural expression that involves PSP names and dimensions in $\mathcal{E}$. We say that e is* implied *from $\mathcal{E}$ if and only if for every value tree T over $\mathcal{D}$ and every embedding M of Q into T, M preserves e. If e can be implied from $\mathcal{E}$ we write $\mathcal{E} \models e$.* □

We provide now a set of inference rules for expressions and we show that it is sound and complete. Let A, B, and C be distinct dimensions of $\mathcal{D}$ and p, p_1, and p_2 be distinct PSP names. We use the symbol $\vdash$ to denote that the expression(s) on its left hand side produce the expression on its right hand side. The absence of expressions in the left hand side of $\vdash$ denotes trivially an inference rule (an axiom). Figure 5 shows a set of 12 inference rules.

(IR 1) *a structural expression that involves the expression* $A[p] \vdash R[p] \Rightarrow A[p]$
(IR 2) $A[p] \rightarrow B[p] \vdash A[p] \Rightarrow B[p]$
(IR 3) $A[p] \Rightarrow B[p],\ B[p] \Rightarrow C[p] \vdash A[p] \Rightarrow C[p]$
(IR 4) $A[p] \rightarrow B[p],\ A[p] \Rightarrow C[p] \vdash B[p] \Rightarrow C[p]$
(IR 5) $A[p] \rightarrow B[p],\ C[p] \Rightarrow B[p] \vdash C[p] \Rightarrow A[p]$
(IR 6) $A[p] \Rightarrow B[p],\ \forall X \in \mathcal{D} - \{A, B\}$ $(X[p] \Rightarrow A[p]$ or $B[p] \Rightarrow X[p]) \vdash A[p] \rightarrow B[p]$
(IR 7) $\vdash R[p_1] \equiv R[p_2]$
(IR 8) $A[p_1] \equiv A[p_2],\ A[p_2] \equiv A[p_3] \vdash$ $A[p_1] \equiv A[p_3]$
(IR 9) $A[p_1] \rightarrow B[p_1],\ B[p_1] \equiv B[p_2] \vdash$ $A[p_2] \rightarrow B[p_2]$
(IR 10) $A[p_1] \Rightarrow B[p_1],\ B[p_1] \equiv B[p_2] \vdash$ $A[p_2] \Rightarrow B[p_2]$
(IR 11) $A[p_1] \Rightarrow B[p_1],\ A[p_1] \equiv A[p_2],\ R[p_2] \Rightarrow B[p_2] \vdash$ $A[p_2] \Rightarrow B[p_2]$
(IR 12) $A[p_1] \Rightarrow B[p_1],\ B[p_1] \equiv B[p_2] \vdash A[p_1] \equiv A[p_2]$

Figure 5: A set of inference rules

If a structural expression e belongs to a set of structural expressions $\mathcal{E}$ or can be produced from $\mathcal{E}$ using the inference rules I.R. 1 - I.R. 12 above we say that e can be *inferred* from $\mathcal{E}$ and we write $\mathcal{E} \vdash e$.

EXAMPLE 6.1. *Consider query Q_4 shown in Figure 6(a). Let $\mathcal{E}$ be its set of structural expressions. Clearly, $\mathcal{E} \vdash C[p_2] \Rightarrow A[p_2]$, and $\mathcal{E} \vdash A[p_2] \Rightarrow C[p_2]$. Consider also query Q_5 shown in Figure 6(b) with $\mathcal{E}$ being its set of structural expressions. It is easy to see that $\mathcal{E} \vdash A[p_1] \equiv A[p_2]$.* □

The proposition below shows that the inference rules above can fully characterize the implication of precedence relationships and node sharing expressions.

PROPOSITION 6.1. *Let $\mathcal{E}$ be a set of structural expressions of a query Q, and e be a structural expression that involves PSP names and dimensions in $\mathcal{E}$. If $\mathcal{E} \vdash e$ then $\mathcal{E} \models e$ (the inference rules are* sound*). If $\mathcal{E} \models e$ then $\mathcal{E} \vdash e$ (the inference rules are* complete*).* □

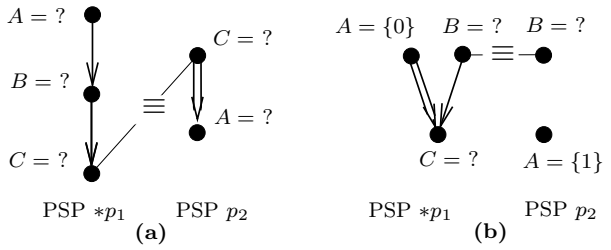

Figure 6: (a) Query Q_4 (b) Query Q_5

A sound set of inference rules is called *redundant* if there is a proper subset of it which is complete. Otherwise, it is called *non-redundant*. We can show that the set of inference rules I.R. 1 to I.R. 12 is non-redundant.

We call *closure* of a set of structural expressions $\mathcal{E}$ the set of structural expressions that can be implied from $\mathcal{E}$. Because of Proposition 6.1 the closure of $\mathcal{E}$ can be computed by first initializing it to $\mathcal{E}$ and then repeatedly adding to it structural expressions produced by the inference rules I.R. 1 - I.R. 12 until no more new structural expressions can be added.

6.2 Unsatisfiable queries

We can identify two notions of query unsatisfiability: a strong one which is independent of the value trees on which the query is evaluated, and a weak one which depends on the dimension graph of the trees on which the query is evaluated. We present each of them in turn and we provide necessary and sufficient conditions for both types of query unsatisfiability.

DEFINITION 6.2. *A query on a dimension set $\mathcal{D}$ is called* unsatisfiable *if its answer is empty on every value tree over $\mathcal{D}$. Otherwise, it is called* satisfiable. □

We qualify this type of unsatisfiability as *strong*, while unsatisfiable queries of this type are called *strongly unsatisfiable*.

EXAMPLE 6.2. *Consider query Q_4 of Figure 6(a). As we saw in example 6.1 the arrows of inferred precedence relationships of Q_4 form a cycle ($A[p_2] \Rightarrow C[p_2] \Rightarrow A[p_2]$). Clearly, this condition implies the unsatisfiability of Q_4. In example 6.1 we showed that the node sharing expression $A[p_1] \equiv A[p_2]$ can be inferred from the structural expressions of Q_5. Clearly, this condition along with the annotating expressions $A[p_1] = \{0\}$ and $A[p_2] = \{1\}$ implies that Q_5 is unsatisfiable.* □

More generally, we can show the following proposition.

PROPOSITION 6.2. *Consider a query Q. Let $\mathcal{N} = \{R[p] \Rightarrow D[p] \mid D$ is an annotated dimension in the PSP p of Q, and $D \neq R\}$. Let $\mathcal{E}$ be the set of structural expressions of Q and $\mathcal{E}' = \mathcal{E} \cup \mathcal{N}$. That is, $\mathcal{E}'$ is the set of structural expressions of Q that includes also ancestor precedence relationships from dimension R to every annotated dimension in every PSP of Q. Let also closure($\mathcal{E}'$) denote the closure of $\mathcal{E}'$. Query Q is unsatisfiable if and only if one of the following conditions holds:*

(a) The double arrows (ancestor precedence relationships) form a cycle in closure($\mathcal{E}'$).

(b) There are two annotating expressions $D[p_1] = V_1$ and $D[p_2] = V_2$ in Q such that $V_1 \neq ?$, $V_2 \neq ?$, $V_1 \cap V_2 = \emptyset$, and $D[p_1] \equiv D[p_2]$ belongs to closure($\mathcal{E}'$). □

A query is defined on a dimension set but it is to be evaluated on a value tree which has a dimension graph. There can be multiple value trees that underlie the same dimension graph. The weak notion of unsatisfiability is defined with respect to a dimension graph.

DEFINITION 6.3. *Let $\mathcal{G}$ be a dimension graph on a dimension set D. A query on $\mathcal{D}$ is* unsatisfiable with respect to $\mathcal{G}$ *if its answer is empty on every value tree underlying $\mathcal{G}$.* □

Given a dimension graph, we qualify this type of unsatisfiability as *weak*, while unsatisfiable queries of this type are called *weakly unsatisfiable*. A weakly unsatisfiable query is also strongly unsatisfiable. The opposite is not necessarily true.

PROPOSITION 6.3. *Let Q be a query on a dimension set $\mathcal{D}$, and $\mathcal{G}$ be a dimension graph on $\mathcal{D}$. Query Q is satisfiable with respect to $\mathcal{G}$ if and only if there is a dimension tree of Q on $\mathcal{G}$.* □

7. EXPERIMENTAL EVALUATION

We experimentally evaluated our approach on a prototype system. We used a set of ten synthetic value trees (with three random partitions each), and we measured the execution time for evaluating random queries. We implemented and compared the following two query evaluation approaches:

$A1$: Queries are formed on dimension graphs. Every query is checked for satisfiability (strong and weak). If it is satisfiable, it is evaluated on the corresponding value tree. Otherwise, its evaluation stops and an empty answer is returned. This is the approach suggested in this paper.

$A2$: Queries are formed directly using sets of branching path expressions that involve values from value trees. Given these sets, the system generates all the possible orderings of the values that respect the parent/child and ancestor/descendant relationships specified in the branching path expressions. Each one of these orderings corresponds to a single branching path expression to be evaluated on a value tree. Note that the approach $A2$ does not exploit dimension graphs for the evaluation of queries.

We carried out three different types of experiments to study the differences in the execution time of random queries for the two query evaluation approaches. For every measure point, 10 queries per partition were generated. The recorded execution time per point is the average execution time for the involved queries.

Varying the size of a query. We measured the execution time varying the percentage of arrows for different numbers of annotated dimensions in queries. The percentage of arrows is the ratio of the number of arrows to the total number of possible arrows in the queries. In Figure 7, we present the results obtained for queries with 2 and 3 PSPs, having 2 and 4 annotated dimensions, varying the percentage of arrows. The number of dimensions is fixed to 30. In each query, 50% of the arrows were single (child relationships) and 50% were double (descendant relationships).

For both approaches, as the percentage of arrows increases, the execution time drops. This is explained by the fact that,

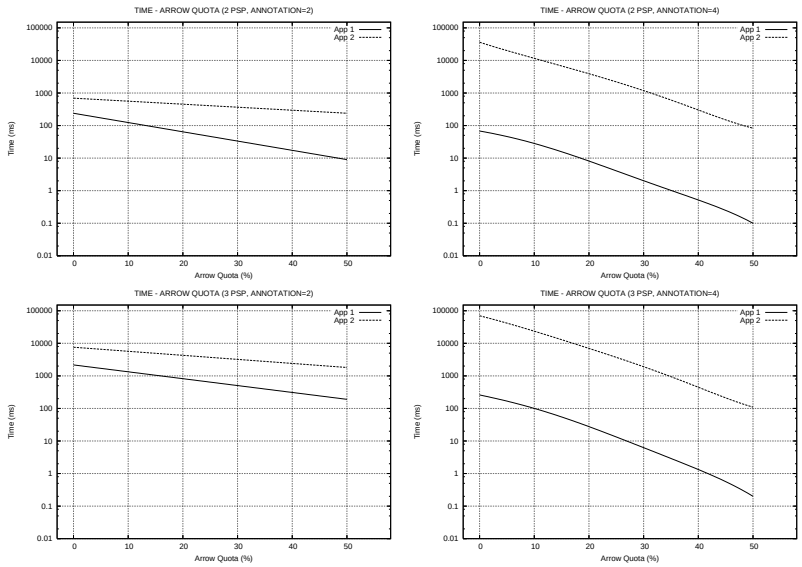

Figure 7: Exec. time vs. % arrows for different numbers of annotated dimensions in queries

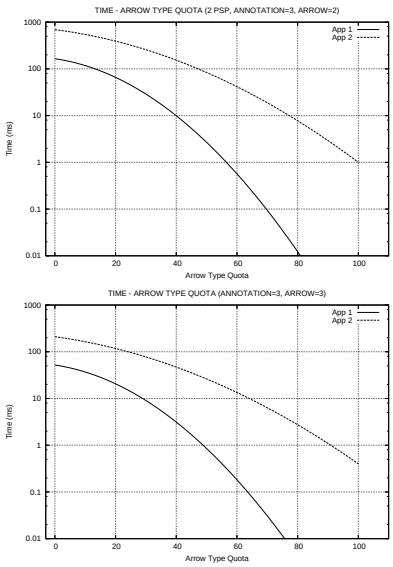

Figure 8: Exec. time vs. % single arrows

as the number of arrows increases, fewer branching path expressions are generated by both approaches to be matched on the value tree.

For queries with 2 PSPs and 4 annotated dimensions, the approach $A1$ outperforms $A2$ by more than 2 orders of magnitude.

Varying the type of arrows in the query. We measured the execution time varying the percentage of single arrows in the total number of arrows in queries for different pairs of numbers of annotated dimensions and arrows.

Figure 8 presents the results obtained for queries with 2 PSPs having (a) 3 annotated dimensions and 2 arrows, and (b) 3 annotated dimensions and 3 arrows. In any case, the approach $A1$ outperforms $A2$ since it is able to exploit the dimension graph to detect unsatisfiable PSPs, and to reduce the number of branching path expressions generated.

8. CONCLUSION

We have addressed the problem of efficiently querying tree-structured data when their structure is not fully available and the problem of querying in an integrated way different tree-structured data sources with structural differences. We have introduced a query language that allows full, partial, or no specification of structural constraints. By exploiting semantic information and summarized structural information of the data sources we have suggested efficient solutions to the aforementioned problems. Our approach bridges the gap between keyword search based approaches on the one side and fully specified tree pattern approaches on the other side.

Our future work includes extending our query language and studying ad-hoc optimization techniques for it.

9. REFERENCES

[1] W3C, http://www.w3c.org.

[2] XML Path Language (XPath). W3C, 2003-2005, http://www.w3c.org/TR/xpath20/.

[3] XML Query (XQuery). World Wide Web Consortium site, The Architecture Domain. W3C, 2003-2005, http://www.w3.org/XML/Query.

[4] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[5] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *Proc. of the EDBT Conf.*, 2002.

[6] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In *Proc. of the ICDT Conf.*, 1997.

[7] A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management*. Addison Wesley, 2003.

[8] Z. Chen, H. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proc. of the VLDB Conf.*, 2003.

[9] V. Christophides, S. Cluet, and J. Simeon. On Wrapping Query Languages and Efficient XML Integration. In *Proc. of the ACM SIGMOD Conf.*, 2000.

[10] S. Cluet, P. Veltri, and D. Vodislav. Views in a Large Scale XML Repository. In *Proc. of VLDB Conf.*, 2001.

[11] S. Cohen, J. Mamou, Y. Kanza and Y. Sagiv. XSearch: A Semantic Search Engine for XML. In *Proc. of the VLDB Conf.*, 2003.

[12] T. Dalamagas, D. Theodoratos, A. Koufopoulos, and V. Oria. Evaluation of Queries on Tree-structured Data Using Dimension Graphs. In *Proc. of IDEAS Symp.*, 2005.

[13] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. In *Proc. of the ICDE Conf.*, 2003.

[14] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *JIIS*, 8(2), 1997.

[15] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of the VLDB Conf.*, 1997.

[16] Y. Lis, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *Proc. of the VLDB Conf.*, 2004.

[17] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries Over Heterogeneous Data Sources. In *Proc. of the VLDB Conf.*, 2001.

[18] N. Polyzotis and M. Garofalakis. Statistical Synopses for Graph-structured XML Databases. In *Proc. of the ACM SIGMOD Conf.*, 2002.

[19] D. Theodoratos and T. Dalamagas. Querying Tree-structured Data Using Dimension Graphs. In *Proc. of the CAiSE Conf.*, 2005.