Evaluation of Queries on Tree-Structured Data using Dimension Graphs*

Theodore Dalamagas
National Technical University of Athens
Athens, GR 15773
dalamag@dblab.ece.ntua.gr

Dimitri Theodoratos
New Jersey Institute of Technology
Newark, NJ 07102
dth@homer.njit.edu

Antonis Koufopoulos
National Technical University of Athens
Athens, GR 15773
akoufop@dblab.ece.ntua.gr

Vincent Oria
New Jersey Institute of Technology
Newark, NJ 07102
oria@homer.njit.edu

Abstract

The recent proliferation of XML-based standards and technologies for managing data on the Web demonstrates the need for effective and efficient management of tree-structured data. Querying tree-structured data is a challenging issue due to the diversity of the structural aspect in the same or in different trees. In this paper, we show how to evaluate queries on tree-structured data, called value trees. The formulation of these queries does not depend on the structure of a particular value tree. Our approach exploits semantic information provided by dimension graphs. Dimension graphs are semantically rich constructs that abstract the structural information of the value trees. We show how dimension graphs can be used to query efficiently value trees in the presence of structural differences and irregularities. Value trees and their dimension graphs are represented as XML documents. We present a method for transforming queries to XPath expressions to be evaluated on the XML documents. We also provide conditions for identifying strongly and weakly unsatisfiable queries. Finally, we conducted various experiments to compare our method for evaluating queries with one that does not exploit dimension graphs. Our results demonstrate the superiority of our approach.

1. Introduction

The recent proliferation of XML-based standards and technologies for managing data on the Web demonstrates the need for effective and efficient management of tree-structured data. Examples of tree-structured data include XML-based repositories, e-commerce product catalogs, taxonomies of thematic categories, concept hierarchies, etc. Even if data is not stored natively in tree structures, export mechanisms make data publicly available in tree structures to enable its automatic processing by programs, scripts, and agents on the Web [1]. The XML language [2] is nowadays the standard data exchange format on the Web for tree-structured data.

Querying capabilities on tree-structured data are provided mainly using queries based on path expressions. A path expression allows the user to navigate through tree-structured data, bind nodes to variables, and identify the part of the tree that satisfies her information need. XPath [3] is a language that uses path expressions to navigate through the tree structure of an XML document. XPath lies at the core of W3C language proposals for XML querying and transformation (e.g. XQuery [4]).

Querying tree-structured data is a challenging issue due to the diversity of the structural aspect in the same or in different trees. For example, querying data sources that use tree structures to organize their data usually need to resolve, besides name mismatches, structural differences and structural irregularities. These can occur in tree structures even for a single knowledge domain. Name mismatches appear because tree structures lack semantic information. For example, `palmtop` devices in one product catalog might be referred to as `handheld` in another catalog. In this paper, we do not focus on this issue and we assume that it is resolved using well-known schema matching techniques [5]. Structural differences and, far more important, structural irregularities appear because of the different possible ways of organizing the same data in tree structures. For example, a structural differ-

*Work supported in part by PYTHAGORAS EPEAEK II programme, EU and Greek Ministry of Education, co-funded by the European Social Fund (75%) and National Resources (25%).

ence exists when a category appears in a product catalog but does not appear in another. A structural irregularity appears when, for instance, a product catalog for notebooks classifies new, SONY notebooks with 10in display in the path `/notebooks/new/Sony/10in`, while another catalog classifies the same products in the path `/Sony/notebooks/10in/new`. Similar problems appear when querying tree-structured data from a single data source. For example, a single XML document may include the element sequence `condition` (e.g. `new`), `brand` (e.g. `Sony`), `pc_type` (e.g. `notebooks`) in one part, but `pc_type, brand, condition` in another part. Such irregularities raise difficulties for querying tree-structured data since query formulation is strictly dependent on the structure. For instance, the user cannot easily form a query to retrieve data without explicitly specifying a sequence for elements `condition`, `brand` and `pc_type`.

A naive approach to cope with these difficulties when querying tree-structured data is to generate different versions of the initial query, by considering different subsets of nodes involved in its path expressions and their different orderings. Clearly, this is not an efficient approach due to the large number of queries that need to be generated, most of which would probably return an empty answer.

In [6], we proposed a method to query tree-structured data, called value trees, by exploiting semantic information. We introduced dimensions which are sets of semantically related nodes in tree structures. Based on dimensions, we defined 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. We also designed in [6] a query language which is not restricted by the structure of the value trees. The user can optionally specify parent-child and/or ancestor-descendent relationships between dimensions in a query.

Contribution. In this paper, we focus on the evaluation of queries on value trees using dimension graphs. Value trees and dimensions are encoded as XML documents. We show how dimension graphs can be used to query efficiently value trees in the presence of structural differences and irregularities. The main contributions of this paper are the following:

- We show how value trees and dimensions can be encoded as XML documents. Queries are not cast on the structure of a specific value tree, since they are issued on its dimensions described in the XML document.
- We present a method for transforming queries to XPath expressions to be evaluated on the XML documents. The XPath expressions are generated using a dimension graph to determine orderings of dimensions that can possibly generate non-empty answers.
- We introduce the concepts of strong and weak query unsatisfiability to identify queries that always produce empty answers. Strong unsatisfiability is detected before the generation of XPath expressions from the dimension graph and prevents the execution of a query. Weak unsatisfiability is detected during the generation of XPath expressions from the dimension graph. This avoids accessing the value tree, which is usually much larger than its dimension graph.
- We present a prototype system that implements our approach. Our system is built on top of an XQuery engine, thus potentially taking advantage of the optimization techniques under development for XQuery language.
- Finally, using our prototype system, we carry 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.

Outline. The rest of the paper is organized as follows. The next section discusses related work. In Section 3, we present the notions of value trees and dimension graphs, and the query language. Section 4 shows how queries are evaluated by generating XPath expressions on XML documents encoding value trees and dimensions. It also provides conditions for checking both type of query unsatisfiability. In Section 5 we describe the architecture of our system. Section 6 presents the experimental evaluation of our approach. We conclude in Section7 and suggest future work.

2. Related Work

Schema-based descriptions for data with little or no apparent structure have also been suggested for semistructured databases. For example, dataguides, which are structural summaries for semistructured data, are introduced in [7]. Statistical synopses for graph-structured XML databases are suggested in [8]. In [9], graph schemas are introduced to formulate, optimize and decompose 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.

Relevant to our work are also techniques where schema descriptions are automatically extracted from local data sources [10, 11]. Contrary to our approach, these papers do not deal with query evaluation.

Query relaxation can also be exploited to search for answers in structrured data. Instead of reformulating the initial query, relaxing techniques can be used to change its form. Tree pattern relaxation methods are presented, for example, in [12]. However, such approaches return approximate and not exact answers.

Finally, many systems support query evaluation on tree-structured data using a predefined global structure and defining mapping rules between this structure and the local structures used in the sources [13, 14, 15, 16]. Our approach can support integration of tree-structured data through the formulation of queries on dimensions spanning all the local value trees. Therefore, it does not require the manual definition of hard-coded mapping rules between the virtual tree structure and the local structures.

3. Data Model and Query Language

In this section we briefly present our data model for tree-structured data and our query language initially introduced in [6].

3.1. Value Trees and Dimensions Graphs

We assume a set of values V that includes a special value r. 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 a set of semantically related values. For instance, `Brand` can be a dimension that includes values `IBM, HP`, and `Mac`. Since the names of the dimensions are distinct we use them to identity the dimensions of $\mathcal{D}$. For the needs of this paper, we assume a single fixed dimension set $\mathcal{D}$.

Definition 3.1 A *value tree* 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.
(c) There are no two nodes on a path in T labeled by values that belong to the same dimension in $\mathcal{D}$.
(d) There are no two sibling nodes in T labeled by the same value. □

Figure 1 shows an example value tree T of a site that sells computers on-line. The same value may label different nodes in a value tree. For instance, value `New`, a value of dimension `condition`, labels two different nodes in T. Figure 1 also shows the grouping of values into different dimensions, e.g. `pc_type, condition, brand`, etc. The partitioning of values in T into dimensions is shown by closed dotted lines labeled by dimensions. The same dimension might label multiple closed dotted lines in T. In this case, this dimension comprises all the nodes surrounded by these closed dotted lines. For instance, the values `Dell, Sony, HP, IBM` and `Mac` belong to the same dimension `brand`. From a structural point of view, the information in T is organized differently under nodes `Notebooks`, `Desktops`, and `PDAs`.

The values of some dimension may not be children or descendents of any value of some other dimension in a value tree. For instance, no value of dimension `pc_type` in the value tree T of Figure 1 is a child or descendent of a value of any of the other dimensions in T 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.
(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* $\mathcal{G}$. □

The dimension graph of a value tree may have cycles. Note that cycles may also trivially involve only two dimensions in a dimension graph.

Figure 2 shows the dimension graph of the value tree T of Figure 1. A trivial cycle is shown in Figure 2 with a double headed edge (the edge between dimensions `condition` and `brand`).

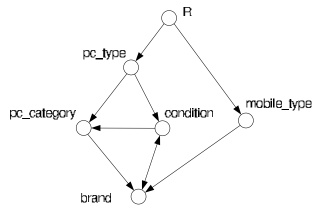

Figure 2. Dimension graph $\mathcal{G}$ of value tree T.

Dimension graphs can be automatically extracted from value trees and abstract their structural information. As we show in subsequent sections, they support the formulation of the queries, and they help the evaluation of queries on value trees and the detection of unsatisfiable queries.

3.2. Queries and Query Answers

Roughly speaking, a user poses a query by annotating some dimensions in a dimension graph with permissible sets of values. The answer comprises root-to-leaf paths on the underlying value tree that involve one value from each of these value sets. An central feature of the language is that the user has the choice of not specifying or partially specifying parent-child and ancestor-descendant relationships between the annotated dimensions in a query.

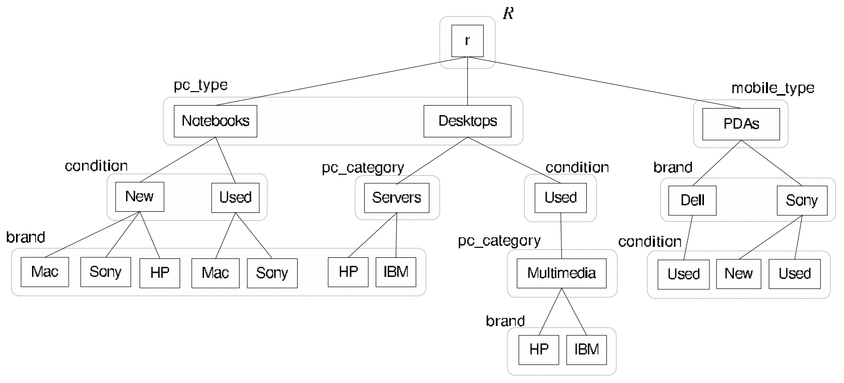

Figure 1. Value tree T

Definition 3.3 Let $\mathcal{D}$ be a dimension set. A *query* Q on $\mathcal{D}$ is a pair $(\mathcal{A}, \mathcal{P})$, where:

(a) $\mathcal{A}$ is a set of expressions of the from $D_i = A_i$, where D_i is a dimension in $\mathcal{G}$ different than R, and A_i is a set of values of dimension D_i or a question mark ("?"). If $D_i = A_i$ belongs to $\mathcal{A}$ we say that D_i is *annotated* in Q, and A_i is called *annotation* of D_i in Q. Without explicitly mention it in $\mathcal{A}$, dimension R is assumed to be annotated with the singleton $\{r\}$. A dimension can be annotated only once in a query.

(b) $\mathcal{P}$ is a set of *precedence relationships* which are expressions of the form $D_i \rightarrow D_j$ or $D_i \Rightarrow D_j$, where D_i and D_j are annotated dimensions of Q.

Sets $\mathcal{A}$ and $\mathcal{P}$ can be empty. $\square$

When a dimension graph $\mathcal{G}$ is given, we can graphically represent a query $Q = (\mathcal{A}, \mathcal{P})$ on $\mathcal{G}$ by labeling its nodes by their annotations in $\mathcal{A}$ and by adding to it a single (resp. double) arrow from node D_i to node D_j for every precedence relationship $D_i \rightarrow D_j$ (resp. $D_i \Rightarrow D_j$) in $\mathcal{P}$. Note that arrows are different than directed edges. The unqualified word "arrow" refers indiscreetly to a single or double arrow.

Consider, for instance, the dimension graph $\mathcal{G}$ of Figure 2. Figure 3 shows the graphical representation of query $Q_1 = (\mathcal{A}, \mathcal{P})$, where $\mathcal{A} = \{\texttt{pc\_type} =?, \texttt{brand} = \{\texttt{Sony}, \texttt{IBM}\}, \texttt{condition} = \{\texttt{used}\}\}$ and $\mathcal{P} = \{\texttt{pc\_type} \Rightarrow \texttt{brand}\}$. Annotated nodes are shown in the figures with black circles. Precedence relationships are shown with single or double arrows from one node to another. For instance, the double arrow from node `pc_type` to node `brand` denotes the precedence relationship in $\mathcal{P}$. In the following we often identify a query with its graphical representation.

The answer of a query on a value tree T is a set of root-to-leaf paths in T compactly represented as a subtree of T.

Definition 3.4 Let $\mathcal{G}$ be a dimension graph of a value tree T over a dimension set $\mathcal{D}$, and Q be a query on $\mathcal{G}$. The *answer* of Q on T is the maximal[1] subtree T' of T such that:

(a) T' and T have the same root r.

(b) Every leaf node of T' is a leaf node of T.

(c) Every path from the root to a leaf node in T' includes one value from every value set annotating a node in Q.

(d) Every path from the root to a leaf node in T' includes one value from every dimension annotated with a question mark in Q.
Therefore, for every annotated node (with a value set or a question mark) in Q, there is one value for the corresponding dimension appearing in every path from the root to a leaf node in T'.

(e) For every path p from the root to a leaf node in T', and for every precedence relationship $D_i \rightarrow D_j$ (resp. $D_i \Rightarrow D_j$) in Q, the value for D_j is a child (resp. descendent) of the value for D_i in p.

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

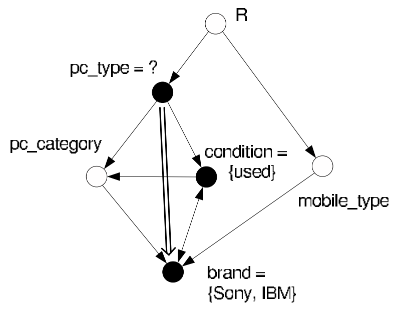

Figure 3. Graphical Representation of Query Q_1

Annotating a node with a "?" in a query is different than not annotating this node at all. In contrast to a non-annotated node, a node that is annotated with a "?" places a

[1]Maximality is meant with respect to the number of nodes or edges.

value of the corresponding dimension in every root-to-leaf path in the answer of the query.

Consider the query Q_1 graphically shown in Figure 3 on the dimension graph $\mathcal{G}$ of Figure 2, Consider also the value tree T of Figure 1 that underlies $\mathcal{G}$. Figure 4 shows the answer T' of Q_1 on T. It also shows the dimensions of the values in T' (which are not part of the answer).

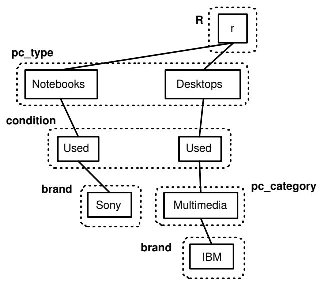

Figure 4. The answer of query Q_1 on value tree T and the dimensions of its values.

4. Query evaluation

In this section we deal with the evaluation of the queries. A query is evaluated on a value tree by generating (possibly several) simple path expressions. These simple path expressions are in turn evaluated on the value trees and their results are composed to form the answer of the query. We use the syntax of XPath for simple path expressions. Thus, we first show, in this section, how XPath expressions are generated. We also introduce two types of unsatisfiability for queries, a strong and a weak one, and we provide necessary and sufficient conditions for a query to be unsatisfiable with respect to each of the two types of unsatisfiability. Detecting the unsatisfiability of a query stops its evaluation at an early stage or before even it starts, and saves accessing the value tree to compute an empty answer.

4.1. XPath expression generation

The generation of XPath expressions is based on the concept of answer path of a query on a dimension graph.

Definition 4.1 Let Q be a query and $\mathcal{G}$ be a dimension graph on a dimension set $\mathcal{D}$. An *answer path* of Q on $\mathcal{G}$ is a path p in $\mathcal{G}$ from the root of $\mathcal{G}$ such that:

(a) All the annotated dimensions in Q are on p, and p ends on an annotated dimension of Q.

(b) If there is a precedence relationship $D_i \rightarrow D_j$ (resp. $D_i \Rightarrow D_j$) in Q, then D_j is a child (resp. descendent) of D_i in p. $\square$

Consider, for instance, the query Q_1 shown in Figure 3 and the dimension graph $\mathcal{G}$ of Figure 2. The answer paths of Q_1 on $\mathcal{G}$ are:

R, `pc_type, condition, brand`
R, `pc_type, condition, pc_category, brand`
R, `pc_type, pc_category, brand, condition`

The answer paths of a query on a dimension graph do not depend on the actual annotations of the nodes in the query, but do depend on the annotated nodes. The presence of an annotated dimension in a query, even if it is annotated by a ?, forces all the answer paths of the query on the dimension graph to include this annotated dimension.

There are two common approaches for implementing membership of values to dimensions: by maintaining a mapping from dimensions to sets of values, or by maintaining a mapping from values to dimensions. We chose the second one in order to support the efficient evaluation of predicates of the form $D = ?$ (the node is labeled by some value of dimension D) at a node of the value tree. Thus, we assume that the dimension of the label of a node is stored with the label or is directly available at the node in a value tree.

In order to enable the use of XPath we introduce an XML document representation of value trees and their dimensions. There is a one-to-one correspondence between the nodes of a value tree and the elements of its corresponding XML document: a node v of dimension D in the value tree corresponds to an element D of the XML document whose text is v. The structure of the value tree is preserved in the XML document. Figure 5 illustrates part of the XML document for the value tree of Figure 1.

```
<R> r
  <pc_type> Notebooks
    <condition> New
      <brand> Sony </brand>
      <brand> HP </brand>
        <brand> Mac </brand>
    </condition>
    <condition> Used
      <brand> Mac </brand>
      <brand> Sony </brand>
    </condition>
  <pc_type> Desktops
    <pc_categoty> Servers
      <brand> HP </brand>
  ...
  </mobile_type>
</R>
```

Figure 5. Part of the XML document representation for the value tree T and its dimensions

The fragment of XPath that we consider here involves node tests, the child axis (/), the descendant axis (//), predicates, the symbol '.' that denotes the current node, and the function `text()` that returns the text of the current element. The predicates are specified between square brackets '[' and ']' following a node test and may involve the Boolean connective `or`. We generate one XPath expression for every answer path of a query on a dimension graph. This process is described below. Let $R, D_1, \ldots, D_k$ be an answer path of a query Q. The corresponding XPath expression has the form $\mathtt{R}[. = \text{“r”}]/\mathtt{t_1}/\ldots/\mathtt{t_k}/\mathtt{text}()$, where for $i = 1, \ldots, k$:

$$t_i = \begin{cases} \mathtt{D}[. = \text{“}\mathtt{v_1}\text{” } \mathtt{or} \ \ldots \ \mathtt{or} \ . = \text{“}\mathtt{v_m}\text{”}] \\ \text{if } D_i \text{ is annotated with the value set } \{v_1, \ldots, v_m\} \\ \mathtt{D}[. = \text{“}\mathtt{v_1}\text{” } \mathtt{or} \ \ldots \ \mathtt{or} \ . = \text{“}\mathtt{v_n}\text{”}] \\ \text{if } D_i \text{ is annotated with a “?” or if } D_i \text{ is not annotated,} \\ \text{and } \{v_1, \ldots, v_n\} \text{ are all the values of dimension } D_i. \end{cases}$$

Consider the answer paths of query Q_1 on dimension graph $\mathcal{G}$ shown above. The corresponding XPath expressions are:

$\mathtt{R}[. = \text{“}\mathtt{r}\text{”}]/\mathtt{pc\_type}/\mathtt{condition}[. = \text{“}\mathtt{Used}\text{”}]/$
$\mathtt{brand}[. = \text{“}\mathtt{Sony}\text{” } \mathtt{or} \ . = \text{“}\mathtt{IBM}\text{”}]/\mathtt{text}()$

$\mathtt{R}[. = \text{“}\mathtt{r}\text{”}]/\mathtt{pc\_type}/\mathtt{condition}[. = \text{“}\mathtt{Used}\text{”}]/$
$\mathtt{pc\_category}/\mathtt{brand}[. = \text{“}\mathtt{Sony}\text{”} \mathtt{or} \ . = \text{“}\mathtt{IBM}\text{”}]/\mathtt{text}()$

$\mathtt{R}[. = \text{“}\mathtt{r}\text{”}]/\mathtt{pc\_type}/\mathtt{pc\_category}/\mathtt{brand}[. = \text{“}\mathtt{Sony}\text{”}$
$\mathtt{or} \ . = \text{“}\mathtt{IBM}\text{”}]/\mathtt{condition}[. = \text{“}\mathtt{Used}\text{”}]/\mathtt{text}()$

The result of an XPath expression e on a value tree T is a (possibly empty) set N of nodes in T. We use XQuery to compute the maximal[2] subtree of T that involves a node in N in every one of its root-to-leaf paths. This subtree is a value tree, and we denoted it $res(e)$. Let $e_1, \ldots, e_k$ are the XPath expressions that correspond to all the answer paths of Q on the dimension graph of T. The answer of Q on T is the value tree obtained by merging all the common paths from the root of the value trees $res(e_i),\ i = 1, \ldots, k$.

4.2. Strongly and Weakly Unsatisfiable Queries

The notion of strong unsatisfiability is independent of the value trees (and consequently of their dimension graphs) on which the query is evaluated.

Definition 4.2 A query on a dimension set $\mathcal{D}$ is called *strongly unsatisfiable* if and only if its answer is empty on every value tree over $\mathcal{D}$. $\square$

Example 4.1 Consider the query $Q_2 = (\mathcal{A}, \mathcal{P})$, where $\mathcal{A} = \{\mathtt{brand} =?, \ \mathtt{condition} =?, \ \mathtt{pc\_category} =?\}$ and $\mathcal{P} = \{\mathtt{condition} \rightarrow \mathtt{brand}, \ \mathtt{pc\_category} \rightarrow \mathtt{brand}\}$. Query Q_2 is shown in

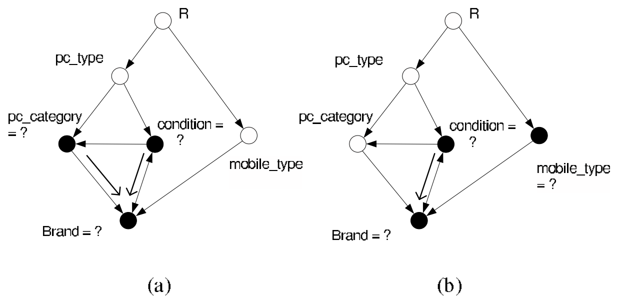

Figure 6. Graphical Representation of (a) Query Q_2, (b) Query Q_3

Figure 6(a). Clearly, no path on any value tree can satisfy both parent-child precedence relationships of this query. Therefore, the answer of Q_2 is empty on any value tree, and Q_2 is a strongly unsatisfable query. $\square$

More generally, we can prove the following proposition which fully characterizes strong unsatisfiability.

Proposition 4.1 A query Q is strongly unsatisfiable if and only if one of the following conditions holds:

(a) The arrows (single and/or double) in Q form a directed cycle.

(b) There are precedence relationships $D \rightarrow D_i$ and $D \rightarrow D_j$ or precedence relationships $D_i \rightarrow D$ and $D_j \rightarrow D$ in Q ($D_i \neq D_j$). $\square$

Based on proposition 4.3, strong unsatisfiability of a query Q can be detected by simply examining the set of precedence relationships of Q. Therefore, the evaluation of a strongly unsatisfiable query stops before it reaches the dimension graph of a value tree.

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

A strongly unsatisfiable query is also weakly unsatisfiable with respect to any dimension graph. In contrast, a weakly unsatisfiable query with respect to a dimension graph is not necessarily strongly unsatisfiable.

Example 4.2 Consider the query $Q_3 = (\mathcal{A}, \mathcal{P})$, where $\mathcal{A} = \{\mathtt{brand} =?, \ \mathtt{condition} =?,$

[2]Maximality is meant with respect to the number of nodes or edges

$\texttt{mobile\_type} = ?\}$ and $\mathcal{P} = \{\texttt{condition} \rightarrow \texttt{brand}\}$. Query Q_3 is shown in Figure 6(b) on the dimension graph $\mathcal{G}$ of Figure 2. This query is not strongly unsatisfiable. Clearly, there is no root-to-leaf path in $\mathcal{G}$ that involves all the annotated dimensions in Q_3 and satisfies the precedence relationships in Q_3. Therefore, Q_3 is weakly unsatisfiable with respect to $\mathcal{G}$. $\square$

The following proposition provides necessary and sufficient conditions for a query to be weakly unsatisfiable with respect to a dimension graph.

Proposition 4.2 Let Q be a query on a dimension set $\mathcal{D}$, and $\mathcal{G}$ be a dimension graph on $\mathcal{D}$. Query Q is weakly unsatisfiable with respect to $\mathcal{G}$ if and only if there is no answer path of Q on $\mathcal{G}$. $\square$

Based on proposition 4.2, weak unsatisfiability of a query Q on a dimension graph $\mathcal{G}$ can be detected when the query evaluator computes the answer paths of Q on $\mathcal{G}$. Therefore, the evaluation of a weakly unsatisfiable query stops before it reaches the value tree underlying the dimension graph. This saves the big bulk of the workload since the dimension graph is much smaller than the underlying value tree.

5. System description

We have implemented a prototype system to study the effectiveness of our approach for querying tree-structured data while partially declaring the required structure (see Figure 7). The *Partition Manager* takes as input a value tree

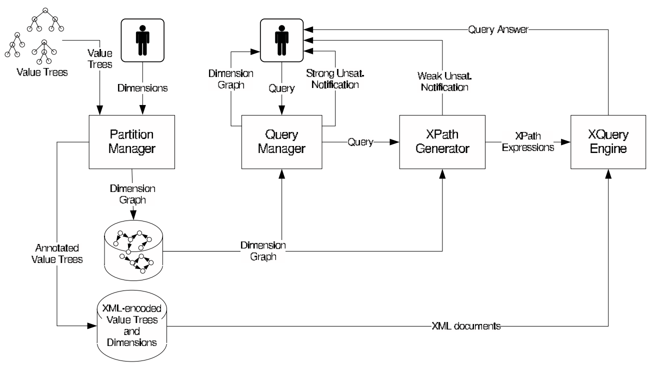

Figure 7. System architecture.

and a mapping of values to dimensions. It produces a new version of the value tree, where all the nodes are annotated with dimensions, encoded as XML document. Also, the Partition Manager extracts the dimension graph of a value tree. The *Query Manager* assists the user to form queries on a value tree. It provides the user with the available set of dimensions and possibly a dimension graph. The user forms a query by annotating dimensions and specifying precedence relationships between annotated dimensions. The module checks and prevents the formulation of a strongly unsatisfiable query. A strongly satisfiable query is subsequently sent to the *Xpath Generator*. The *XPath Generator* takes as input a strongly satisfiable query and a dimension graph and produces the answer paths of the query on the dimension graph and their corresponding XPath expressions. It thus checks whether the query is weakly unsatisfiable in order to prevent its execution on the value tree and to notify the user. If the query is satisfiable, this module exploits the dimension graph of the value tree to produce XPath expressions needed to answer the query. These XPath expressions are then sent to the *XQuery Engine*. The *XQuery Engine*[3] evaluates the XPath expressions on the value tree and sends the answer to the user.

6. Experimental Evaluation

We experimentally evaluated our approach on our prototype system[4]. We used a set of synthetic value trees, and we measured the execution time for evaluating queries.

To produce the data set of synthetic value trees we exploited an XML generator[5] tuned appropriętly to produce value trees encoded as XML documents. The structure of value trees generated was determined by DTDs given as input to the XML generator. Changing randomly the status of element descriptions to 'optional' in DTD rules, the XML generator produces value trees with structural differences and inconsistencies. Ten XML documents were generated for the experiments. After the construction of the value trees, we generated random partitions of their values. Three partitions per value tree (that is 30 dimension graphs) were created for the experiments.

Queries were generated by randomly annotating dimensions in dimension graphs and adding arrows. If k dimensions need to be annotated, the generator selects randomly k dimensions and annotates them with '?', one value, or a set of values with a certain probability given as a parameter. In order to add arrows to the annotated dimension of the query, the generator first creates a fully connected graph, involving only the annotated dimensions. Then, if n arrows need to be created, it removes arrows until n are left. The percentage of single arrows in the total number of arrows in the query is a system parameter and depends on the experiments. 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 sat-

[3]We used Qexo XQuery engine (http://www.qexo.org/)

[4]Experiments were made on an AMD Sempron 2600 PC, 512MB RAM.

[5]http://www.alphaworks.ibm.com/tech/xmlgenerator

isfiable, 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 path expressions (parent/child and ancestor/descendant relationships) 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 path expressions. Each one of these orderings corresponds to a single path expression to be evaluated on a value tree. Note that the approach $A2$ does not exploit dimension graphs for the evaluation of queries.

In order to maintain similar query sets for both approaches, the system transforms a query that involves dimensions into a set of simple path expressions to be matched by the same path of the value tree. Consider, for instance, the query $(\mathcal{A}, \mathcal{P})$, where $\mathcal{A} = \{\texttt{pc\_type} = \{\texttt{Notebooks}\}, \texttt{brand} = \{\texttt{Sony}, \texttt{IBM}\}, \texttt{condition} = \{\texttt{Used}\}\}$ and $\mathcal{P} = \{\texttt{pc\_type} \rightarrow \texttt{brand}\}$. The set of simple path expressions for approach $A2$ is {`r//Notebooks/(Sony|IBM)`, `r//Used`}. The corresponding path expressions to be evaluated on the value tree are `r//Notebooks/(Sony|IBM)//Used` and `r//Used//Notebooks/(Sony|IBM)`.

6.1. Experiments

We carried out three different types of experiments to study the differences in the execution time of the two query evaluation approaches. For every measure point in the x-axis, 10 queries per partition were generated (that is, a total of 30 queries per point for each one of the 10 value trees). The recorded execution time per point is the average execution time.

Varying the size of the queries. We measured the execution time varying the percentage of arrows (i.e. precedence relationships) for different numbers of annotated dimensions in the queries. The percentage of arrows is the ratio of the number of arrows to the total number of possible arrows in the query. Note that a percentage of arrows of 100% means that the arrows and the annotated dimensions of the query form a fully connected graph. In Figure 8, we present the results obtained for queries having 2 to 7 annotated dimensions, varying the percentage of arrows. The y-axis is on a logarithmic scale. The number of dimensions is fixed to 15. In each query, 50% of the arrows were single (parent/child relationships) and 50% were double (ancestor/descendant relationships).

For both approaches, as the percentage of arrows increases, the execution time drops. This is explained by the fact that, as the number of arrows increases, fewer path expressions are generated by both approaches to be matched on the value tree.

As the number of annotated dimensions increases, the execution time in approach $A1$ drops. This is expected, since for a fixed dimension graph, an increase in the number of annotated dimensions reduces the number of possible answer paths (recall that an answer path involves all the annotated dimensions). Therefore, the number of path expressions generated by approach $A1$ to match the values tree is reduced too.

In approach $A2$, as the number of annotated dimensions increases, the query execution time raises significantly for low arrow percentage, but the steepness of the fall of the curve raises too. The curve hits the x-axis closer to 0 as the number of annotations raises. This can be explained as follows. As the number of annotated dimensions increases, the number of possible value orderings increases exponentially. For a fixed set of arrows, this increase results in an increase on the number of path expressions generated. However, for a fixed percentage of arrows, the number of arrows increases too when the number of annotated dimensions increases. As we explained above, increasing the number of arrows reduces the number of path expressions generated. For a fixed percentage of arrows, after a certain threshold number of annotated dimensions, the increase in the number of arrows dominates and the number of generated path expressions drops.

For a low number of annotated dimensions (e.g. 2 or 3), and high percentage of arrows, the approach $A2$ outperforms $A1$. The reason is that the number of possible orderings of the values is low and the path expressions are posed directly on the value trees, without the overhead of generating answer paths and checking for unsatisfiability. However, for a higher number of annotated dimensions, the generation of orderings of values is too costly. In this case, the approach $A1$ outperforms $A2$. For an arrow percentage of 10%, and 6 or 7 annotated dimensions, the approach $A1$ outperforms $A2$ by almost 3 orders of magnitude.

Varying the type of arrows in the queries. We measured the execution time varying the percentage of single arrows in the total number of arrows in the query for different pairs of numbers of annotated dimensions and arrows. In Figure 9, we present the results obtained for queries having (a) 6 annotated dimensions and 4 arrows, (b) 7 annotated dimensions and 4 arrows, and (c) 7 annotated dimensions and 5 arrows.

The higher the percentage of single arrows, the lower the number of possible orderings of values needed for $A2$. This is reflected in the diagram, since there is a drop in the execution time as the percentage of single arrows increases for $A2$. For $A1$, higher percentage of single arrows means (a) higher probability for a generated query to be unsatisfiable, and (b) less answers paths. The reason is that the

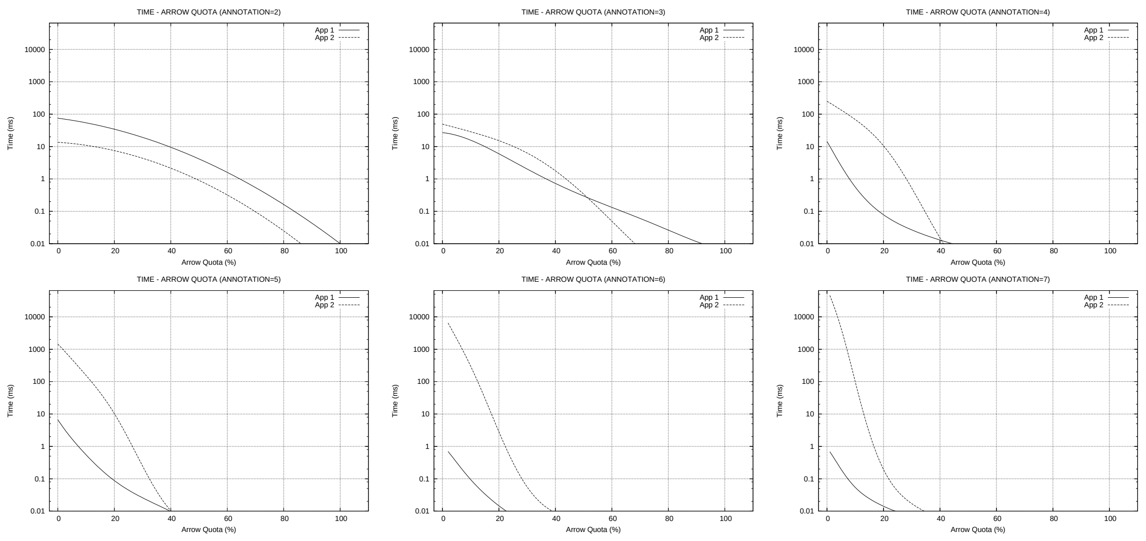

Figure 8. Execution time varying the percentage of arrows for different numbers of annotated dimensions in the query.

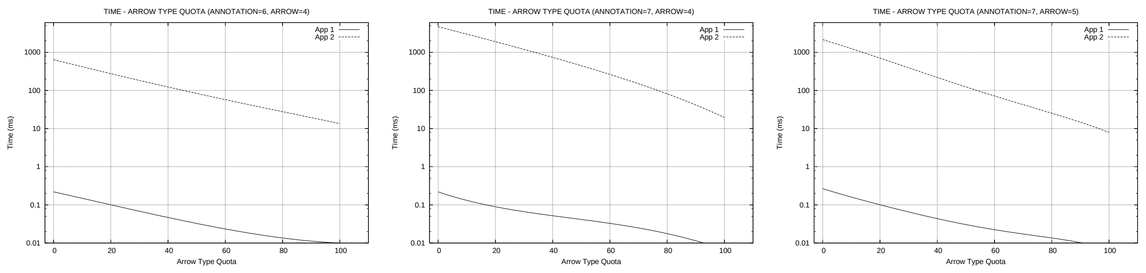

Figure 9. Execution time varying the percentage of single arrows in the total number of arrows in the query, for different pairs of numbers of annotated dimensions and arrows.

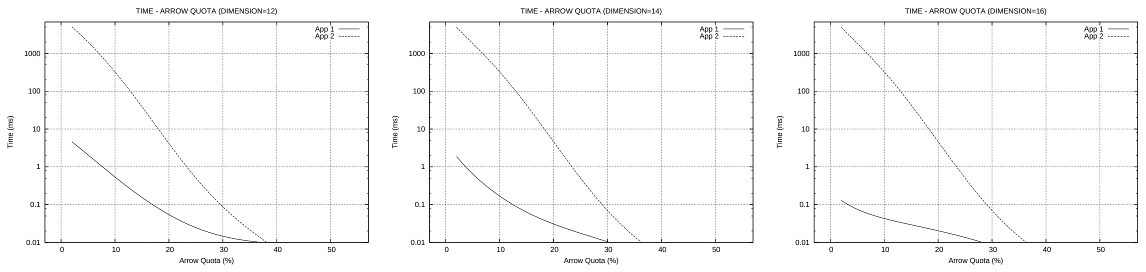

Figure 10. Execution time varying the percentage of arrows for different numbers of dimensions in the dimension graphs.

constraints imposed by single arrows are more restrictive than those of double arrows. This is also reflected in the diagram, since there is a drop in the execution time as the percentage of single arrows increases for $A1$. In any case, the approach $A1$ outperforms $A2$ since it is able to exploit the dimension graph to detect unsatisfiable queries, and to reduce the number of path expressions generated. In most cases, the approach $A1$ outperforms $A2$ by three to four orders of magnitude.

Varying the size of dimension graphs. We calculated the execution time varying the percentage of arrows, for different sizes of dimension graphs. The number of annotated dimensions was fixed for different numbers of dimensions. In Figure 10, we present the results obtained for dimension graphs having 12, 14 and 16 dimensions. In each query, the number of annotated dimensions was fixed to 6 and 50% of the arrows were single ones.

Before we discuss the results, we explain the way we increase the size of a dimension graph. Starting from a fixed value tree, its partition and a set of annotated dimensions, we generate queries by randomly adding arrows between those annotated dimensions. At the next step, a dimension is randomly selected to be split, and produces two new dimensions. The arrows are re-assigned to the new dimensions that contain the annotated values. When the number of dimensions increases, their number of values per dimension decreases on the average. In general, this results in a sparser graph and reduces the number of answer paths of a query. This is reflected in the diagram, since there is a drop in the execution time for $A1$ as the number of dimensions increases.

Note that in this experiment, the execution time of the approach $A2$ remains unaffected from the increase in the number of dimensions, since the queries do not change and the approach $A2$ does not involve dimensions and dimension graphs. For an arrow percentage of 10%, the difference between the execution time of the two approaches changes from almost three to four orders of magnitude.

7. Conclusion

Dimensions are sets of semantically related nodes in a type of tree-structured data, called value trees. Dimension graphs are based on dimensions and are semantically rich constructs that abstract the structural information of the value trees. In this context, we considered queries that are specified on dimensions and are not cast on the structure of a specific value tree. We presented a method to evaluate queries on value trees. We showed how value trees and dimensions can be encoded as XML documents. We designed a method for transforming queries to XPath expressions to be evaluated on the XML documents. We provided conditions for detecting strongly and weakly unsatisfiable queries. These are queries that always produce empty answers when evaluated with respect to a dimension graph. We presented a prototype system that implements our approach on top of an XQuery engine. We carried out experiments to compare our approach to one that does not exploit dimension graphs in the evaluation of queries on value trees. Our results demonstrated the superiority of our approach.

References

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

[2] W3C Consortium, www.w3c.org.

[3] XPath Lang., W3C, www.w3c.org/TR/xpath20/.

[4] XQuery Lang., W3C, www.w3.org/XML/Query.

[5] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *VLDB Journal*, vol. 10, no. 4, 2001.

[6] D. Theodoratos and T. Dalamagas, "Querying tree-structured data using dimension graphs," in *Proc. of the CAiSE'05 Conf.*, Porto, Portugal, 2002.

[7] R. Goldman and J. Widom, "DataGuides: Enabling query formulation and optimization in semistructured databases," in *Proc. of the VLDB'97 Conf., Athens, Greece*, 1997.

[8] N. Polyzotis and M. Garofalakis, "Statistical synopses for graph-structured XML databases," in *Proc. of the ACM SIGMOD'02 Conf., Madison, Wisconsin, USA*, 2002.

[9] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu, "Adding structure to unstructured data," in *Proc. of the ICDT'97 Conf.*, Delphi, Greece, 1997.

[10] M. L. Lee, L. H. Yang, W. Hsu, and X. Yang, "Xclust: Clustering XML schemas for effective intergration," in *Proc. of the CIKM'02 Conf., McLean, USA*, 2002.

[11] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim, "XTRACT: A system for extracting document type descriptors from XML documents," in *Proc. of the ACM SIGMOD'00 Conf., Dallas, USA*, 2000.

[12] S. Amer-Yahia, S. Cho, and D. Srivastava, "Tree pattern relaxation," in *Proc. of the EDBT'02 Conf., Prague, Czech Republic*, 2002.

[13] S. Cluet, P. Veltri, and D. Vodislav, "Views in a large scale XML repository," in *Proc. of the VLDB'01 Conf., Rome, Italy*, 2001.

[14] I. Manolescu, D. Florescu, and D. Kossmann, "Answering XML queries over heterogeneous data sources," in *Proc. of the VLDB'01 Conf., Rome, Italy*, 2001.

[15] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl, "Ontology-based integration of XML Web resources," in *Proc. of the ICSW'02 Conf., Sardinia, Italy*, 2002.

[16] V. Christophides, S. Cluet, and J. Simeon, "On wrapping query languages and efficient XML integration," in *Proc. of the ACM SIGMOD Conf.*, 2000.