Semantic Integration of Tree-Structured Data Using Dimension Graphs*

Theodore Dalamagas[1], Dimitri Theodoratos[2], Antonis Koufopoulos[1], and I-Ting Liu[2]

[1] School of Electr. and Comp. Engineering,
National Technical University of Athens,
Athens, GR 15773
{dalamag,akoufop}@dblab.ece.ntua.gr

[2] Department of Computer Science,
New Jersey Institute of Technology
Newark, NJ 07102
{dth,il2}@cs.njit.edu

Abstract. Nowadays, huge volumes of Web data are organized or exported in tree-structured form. Popular examples of such structures are product catalogs of e-market stores, taxonomies of thematic categories, XML data encodings, etc. Even for a single knowledge domain, name mismatches, structural differences and structural inconsistencies raise difficulties when many data sources need to be integrated and queried in a uniform way. In this paper, we present a method for semantically integrating tree-structured data. We introduce dimensions which are sets of semantically related nodes in tree structures. Based on dimensions, we suggest dimension graphs. Dimension graphs can be automatically extracted from trees and abstract their structural information. They are semantically rich constructs that provide query guidance to pose queries, assist query evaluation and support integration of tree-structured data. We design a query language to query tree-structured data. The language allows full, partial or no specification of the structure of the underlying tree-structured data used to issue queries. Thus, queries in our language are not restricted by the structure of the trees. We provide necessary and sufficient conditions for checking query satisfiability and we present a technique for evaluating satisfiable queries. Finally, we conducted several experiments to compare our method for integrating tree-structured data with one that does not exploit dimension graphs. Our results demonstrate the superiority of our approach.

1 Introduction

Nowadays, huge volumes of data are posted and retrieved through the Web. Despite this vast exchange of information, there is no consistent and strict orga-

* 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%).

nization of data, raising difficulties for its sharing and processing. For the Web to reach its full potential and become a universally accessible platform, it should support effective retrieval and integration of the posted data. Tree structures play an important role in this task, providing a means to organize the information on the Web. Taxonomies of thematic categories, concept hierarchies, e-commerce product catalogs are examples of such structures. The XML language [3] is nowadays the standard data exchange format on the Web for tree-structured data. 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. 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 [13].

Querying capabilities on these structures are provided either through browsing tools or through path expression queries. For the former, the user should navigate among nodes to identify data. For the latter, she should form queries using some of the query languages proposed in the literature (e.g. XPath [4], XQuery [5]). For example, /notebooks/new/ultralight[price<2000] is an XPath expression that will retrieve new, ultralight notebooks that cost less than $2000.

The problem. Under the information integration perspective [24], a challenging issue is to integrate and query in a uniform way many tree-structured data sources. Users should be able to pose a query on a 'global' tree structure. The answer of the query is formed using data retrieved from 'local' data sources. The whole process should be transparent to the user in the sense that she need not know details about the local data sources and the query processing. Even for a single knowledge domain, integrating tree-structured data turns out to be a hard task due to name mismatches, structural differences and structural inconsistencies. Name mismatches appear because tree structures lack semantic information. For example, laptop computers might be referred to as notebooks in one product catalog but as portables 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 [28]. Structural differences and, far more important, structural inconsistencies appear because of the different possible ways of organizing the same data in tree structures. For example, a structural difference exists when a category appears in a product catalog but does not appear in another. A structural inconsistency appears when a product catalog for notebooks classifies new, SONY notebooks with $10''$ display in the path /notebooks/new/SONY/$10''$, while another catalog classifies the same products in the path /SONY/notebooks/$10''$/new. As a result, a path expression query in the form of /notebooks/SONY/new/$10''$ should be reformulated to match the structure of each catalog.

Current tree-structured data query languages (e.g. Xquery) handle this issue in a procedural way, in the sense that the user should explicitly specify structural differences and irregularities as part of the query itself. For example, to identify new, SONY notebooks with $10''$ display as in the previous example, the user should explicitly specify alternate sequences for categories and use disjunctions

in her query. Requiring such a strict specification raises difficulties in forming queries.

A naive approach to cope with structural differences and inconsistencies is to generate different versions of the initial query, considering different subsets of nodes involved in its path expressions and their different orderings. Clearly this is not efficient due to the large number of queries that need to be generated. Instead of reordering the initial query, relaxing techniques can be used to change its form and search for answers in local data sources [8, 19]. For example, query `/new/monitors/CRT/17″` can be relaxed to `/new/monitors//17″`, where '//' denotes ancestor-descendand relationship. The relaxed query asks for new 17″ monitors, without specifying whether these monitors are TFT or CRT, and permits other nodes to appear between nodes `monitor` and `17″`. Nevertheless, such techniques return approximate and not exact answers.

A traditional approach to information integration defines mapping rules between a global structure and the local structures used in the sources [15]. For example, given the rules `(notebooks/Sony)` → `(Sony/notebooks)` and `(new/10″)` → `(10″/new)` for one catalog, query `/notebooks/Sony/new/10″` will become `/Sony/notebooks/10″/new` in order to match its structure. Such approaches require extensive manual effort, since the global schema is difficult to contruct and the rules should be hard-coded in the integration application.

Our approach. In this paper, we suggest a novel approach to the integration of tree-structured data. Tree-structured data provides mainly syntactic and not semantic information. However, there are inherent semantics, for instance, subcategories of a main category in a catalog are usually related under a semantic interpretation given by the author of that particular catalog. Subcategories `notebooks` and `desktops`, for example, indicate that there are certain items that can be characterized with a property that indicates their product type (i.e. notebook or desktop). Our approach captures and exploits such a semantic information for querying tree-structured data (called here value trees). Such a semantic information plays a two-fold way: (a) it becomes part of a query language itself and (b) it provide a means for optimizing queries in tree-structured data.

We introduce the concept of a dimension that groups together semantically related values (nodes). For instance, `IBM, Sony, HP` can be values of dimension `brand`. We assume that the semantic interpretation of the values is available. The nodes of a value tree are partitioned into dimensions. The different dimensions of a value tree are related through precedence relationships incurred by the parent-child and ancestor-descendant relationships of their nodes. We capture these precedence relationships between dimensions of a value tree into the concept of a dimension graph for the value tree. Dimension graphs can be automatically extracted from trees and abstract their structural information. They are semantically rich constructs that provide query guidance to pose queries, assist query evaluation and support integration of tree-structured data.

Query conditions involve dimensions, and thus query formulation is not dependent on the structure of value trees. For instance, the query above asking for

new Sony notebooks with 10″ display would look like: `pc_type = Notebooks`, `brand = Sony`, `condition = New`, `display_size = 10″`, where `pc_type`, `brand`, `condition` and `display_size` are dimensions. No order is a priori specified among values (nodes), unless the user wants to impose a partial or a total order. The system uses the dimension graph of the value tree to identify orderings of the values that can possibly exist in the value tree. Only these value orderings will be used to compute the answer of the query on the value tree. This step of the computation of the query answer is performed before the query evaluation reaches the value tree which is, in general, much larger than its dimension graph. Dimension graphs provide also the means for integrating different data sources. This is achieved through the creation of a 'global dimension graph' which is a merging of the dimension graphs of the data sources. User queries are issued against the global graph and they are then translated into queries on the local dimension graphs where they are evaluated.

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

- We introduce dimensions to record semantic information for the nodes of value trees. We also introduce dimension graphs to capture structural information on value trees. Given a partitioning of the nodes of value trees into dimensions, the dimension graphs for these value trees can be automatically extracted.
- We design a query language for this framework. Queries are not issued directly on value trees but on their dimensions. Therefore, queries are not cast on the structure of a specific value tree. The user can optionally specify parent-child and/or ancestor-descendent relationships between dimensions in a query.
- We show how queries on value trees can be evaluated. In this process, the dimension graph of the value tree plays a two-fold role. First, it allows identifying an unsatisfiable query, that is, a query that does not have an answer on any value tree underlying the dimension graph. Second, if the query is satisfiable, it allows determining those orderings of dimensions that can possibly generate non-empty answers. Thus, dimension graphs prune useless dimension orderings at an early stage of the evaluation of a query. We provide necessary and sufficient conditions for a query to be unsatisfiable, and we show how path expressions to be evaluated on value trees are generated from non-filtered dimension orderings.
- We present a method for integrating different data sources through the creation of a global dimension graph. We show how queries on the global dimension graph can be evaluated first at the global site and then at the data sources.
- We carry out several experiments to compare our approach to one that does not exploit dimension graphs in the integration of tree-structured data sources. Our results demonstrate the superiority of our approach.
- Our approach can be applied to the integration of tree-structured data in various application areas including the integration of product catalogs with

different structures in e-commerce applications or the integration of XML data from similar knowledge domains that conform to different DTDs.

Outline. The rest of the paper is organized as follows. The next section discusses related work. In Section 3, we introduce dimensions and we define dimension graphs for value trees. Section 4 presents the query language used to pose queries on dimension graphs. It also shows how queries can be checked for unsatisfiability and how they are evaluated on the underlying value trees. In Section 5, we show how data sources can be integrated and queried in a uniform way. Section 6 presents the experimental evaluation of our approach. Finally, Section 7 concludes the paper and presents further work.

2 Related Work

Tree-structured data integration has become a popular research issue, especially after the latest increase in the use of XML to encode data. The vast majority of suggested techniques follow traditional information integration methods. Given a predefined virtual structure (i.e. a global schema), mapping rules are defined between the virtual structure and the local structures of the sources. The initial query is posed on the virtual structure and transformed to queries on local structures using the mapping rules.

The Xyleme system [15] copes with the problem of integrating XML data sources by defining and querying views. The user creates a DTD to act as the global schema. Queries are expressed using query pattern trees. Query evaluation exploits mapping rules in the form of path-to-path correspondences. The Agora system [25] integrates relational and tree-structured XML data sources under a global XML schema. Users express a query in the XQuery language over a given global XML schema. The query is translated to an intermediate SQL query, and then to SQL queries on local data sources. In [7], a methodology to integrate XML Web resources is presented. The global schema used to pose OQL queries is a lightweight ontology with object-oriented model primitives. Query evaluation is based on mapping rules in the form of path-to-path correspondences. In [14], an XML integration system based on the YAT model is described. YAT queries on a global schema are translated to an algebraic form based on a set of algebraic operators. The algebraic translations of the initial query are further processed using YAT mapping rules before they are evalutated in the data sources. In [26], an adaptive evaluation technique is presented for querying XML-based electronic catalogs. Given a global DTD, Xpath queries are formed and reformulated to queries on the local catalogs. Query evaluation is based on mapping rules between elements of the global DTD and their representation in local catalogs. A language for querying XML sources is presented in [12]. Queries are issued on a global conceptual schema and translated to local data sources using a pre-defined set of mappings. A detailed survey about general schema integration techniques can be found in [29]. Our approach differs than the aforementioned traditional information integration techniques in that

it does not require the manual definition of hard-coded mapping rules between the virtual tree structure and the local structures. Further, our approach does not use a predefined virtual global schema. In contrast, a global dimension graph is constructed automatically from local data sources.

Relevant to our work are also integration techniques where a global structure is not predefined, but rather is constructed using schema descriptions extracted from local data sources. In [16], global schemas are generated from the semantic integration of conceptual schemas extracted from DTDs of XML data sources. Similarly, XClust [23] generates DTDs to act as glocal schemas, applying clustering methods to detect similar DTDs prior to their integration. Techniques that extract DTDs from collections of XML documents are also presented in [17]. In [9], a grammar-based model based on tree automata is used to integrate DTDs. Contrary to our approach, these papers do not deal with query evaluation.

Schema-based descriptions for data with little or no apparent structure have also been suggested for semistructured databases [6]. Dataguides are introduced in [18]. They are structural summaries for semistructured data, useful for formulating queries, storing statistics about paths and nodes, and enabling query optimization. Statistical synopses for graph-structured XML databases are suggested in [27]. In [11], graph schemas are introduced to formulate, optimize and decompose queries for semistructured data. Database conformity to a graph schema is based on notion of graph simulation. These approaches do not provide a direct solution to the problem of structural inconsistencies and differences in data sources that we address here. Further, they are purely syntactic. In contrast to our approach, they do not exploit semantic information.

Integrating tree-structured data is also a popular issue in e-commerce applications. A system to integrate product classification schemes is presented in [10]. Using the source descriptions, the system generates a glocal schema based on inter-schema and intra-schema relationships determined manually. In [22], a method to integrate e-commerce catalogs is presented. Related categories are organized in term vectors. Membership rules are defined to encode parent/child relationship between categories. The integrated catalog contains all information from the original catalogs, maintaining at the same time their structural information. Facet classification hierarchies [1, 2] also exploit sets of semantically related categories. Facets provide different classification schemes for the same data. In [31], the authors present faceted taxonomies for Web catalogs. They investigate the problem of invalid navigation paths produced after the combination of taxonomies corresponding to different facets. None of these papers suggest query evaluation techniques. Preliminary work on querying tree-structured data using dimension graphs has also been presented in [30]. However, this paper does not consider the problem of integrating tree-structured data sources.

3 Data Model

In this section we present a data model for tree-structured data. We introduce a type of trees, called value trees, to represent tree-structured data. We also

discuss the notion of a dimension, based on which a partitioning can be enforced on value trees.

3.1 Value Trees and Dimensions

We assume a set of values V that includes a special value r. The elements of V are used to build value trees.

Definition 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 sibling nodes in T labeled by the same value. □

Example 1. Figure 1 shows examples of value trees T_1, T_2 and T_3 (for the moment, the dotted labeled rectangles that group the nodes should be ignored). These value trees are parts of taxonomies used to categorize products related to computer equipment. The same value may label multiple nodes in a value tree. For example, value `HP` labels two nodes in T_2. Notice that there are stuctural differences and inconsistencies between value trees T_1, T_2 and T_3, although they refer to the same knowledge domain. For example, there are nodes labeled `Multimedia` or `Servers` in T_2 and T_3, even though no such nodes appear in T_1. Also, a node labeled `Used` is a child of a node labeled `Sony` in T_2, although the opposite holds in T_3. Note that we assume that naming mismatches have been resolved. For instance, nodes labeled by the same value in different trees refer to the same real world concept. □

Values in set V can be grouped to form dimensions. Intuitively, a dimension is a set of semantically related values. For instance, values `Mac`, `Acer` and `Compaq` can be interpreted as values of a dimension `brand`. A semantic interpretation of values is imposed by a user. A dimension can also be seen as a property with values.

Definition 2. *Let V be a set of values that includes a specific value r. A* dimension set *over V is a partition $\mathcal{D}$ of V that includes a set R whose single element is value r. Each element of $\mathcal{D}$ is called* dimension. □

Example 2. Figure 2 shows a dimension set $\mathcal{D}$ and the names of its dimensions. We use these dimensions and the value trees of Figure 1 as a running example in this paper. □

A dimension set also partitions the nodes of a value tree. We are interested in value trees where every path from the root to a leaf involves values from distinct dimensions. To describe this type of value trees we introduce the concept of *tree conformity* with respect to a dimension set.

Definition 3. *Let $\mathcal{D}$ be a dimension set over a value set V. A value tree T* conforms *to $\mathcal{D}$ iff there are no two nodes on a path in T labeled by values that belong to the same dimension in $\mathcal{D}$.* □

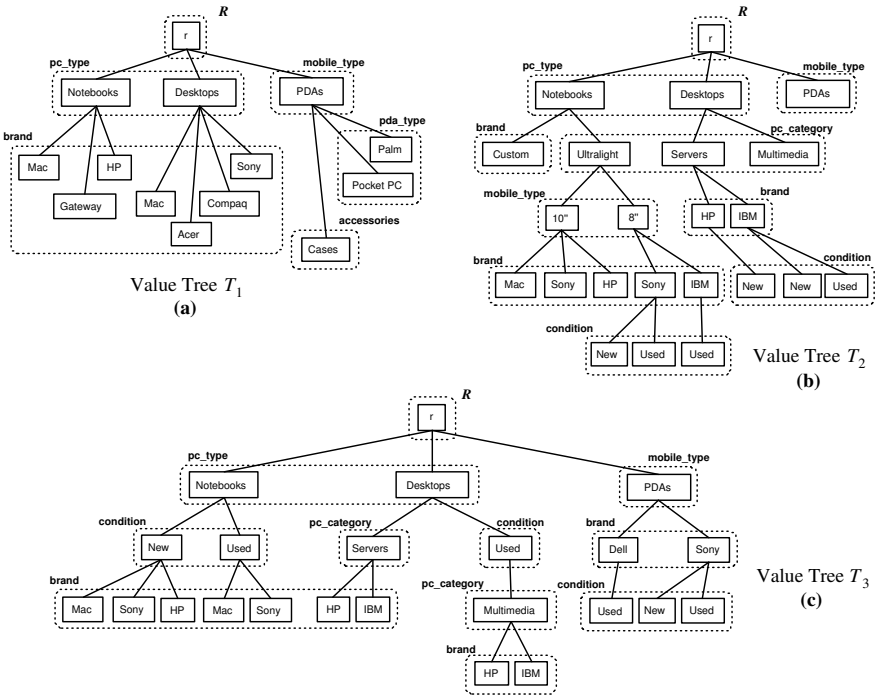


Fig. 1. Value trees T_1, T_2 and T_3

Example 3. Consider, for example, the value trees T_1, T_2 and T_3 of Figure 1. Dotted rectangles labeled by dimensions are used to show the partitioning of nodes into dimensions. The same dimension might label different rectangles in a value tree. In this case, this dimension comprises the nodes confined by all these rectangles. Dimension `pc_type` in T_1 refers to types of personal computers and includes nodes labeled by values `Desktops` and `Notebooks`. Dimension `brand` in T_3 refers to brand names and includes nodes labeled `Mac, Sony, HP, IBM` and `Dell`. All trees T_1, T_2, and T_3 conform to the dimension set $\mathcal{D}$ shown in Figure 2. □

Nodes labeled by values of the same dimension need not be in the same level of a value tree. For example, in T_2, the nodes labeled `10''` and `8''` of dimension `mobile_type` are not in the same level as the node labeled `PDAs` of the same dimension. A value of a dimension may not appear in a value tree. For example, the value `Ultralight` of dimension `pc_category` does not appear in value tree T_3 nor in T_1, although it appears in T_2. Further, a dimension may have no value in a value tree. For instance, no value of `pc_category` appears in T_1.

In the following we assume that a dimension set $\mathcal{D}$ is given and all value trees conform to $\mathcal{D}$.

Dimension Set D = { R, pc_type, brand, mobile_type, pda_type, accessories, pc_category, condition }

Dimensions:
pc_type = { Notebooks, Desktops }
brand = { Mac, Sony, HP, IBM, Gateway, Acer, Compaq }
mobile_type = { PDAs, 10", 8" }
pda_type = { Palm, Pocket_PC }
accessories = { Cases }
pc_category = { Ultralight, Multimedia, Server }
condition = { New, Used }

Fig. 2. A dimension set and its dimensions

3.2 Dimension Graphs

Values of one dimension can label children or descendants of nodes labeled by values of any other dimension in a value tree. However, there are cases where values of one dimension do not label descendants of nodes labeled by values of some other dimension. For example, none of the values `Pocket_PC` and `Palm` of dimension `pda_type` labels a descendant of the nodes labeled by the value `Desktops` or `Notebooks` of dimension `pc_type` in the value tree T_1 of Figure 1. To capture this type of relationship between dimensions in a value tree, we introduce the concept of a dimension graph. Dimension graphs can be automatically extracted from value trees and abstract their structural information. Moreover, they provide semantic query guidance to pose and evaluate queries on value trees (see subsequent sections). Before we give the formal definition of a dimension graph with respect to a value tree, we define dimension graphs as general structures.

Definition 4. *A* dimension graph *over dimension set $\mathcal{D}$ is a directed graph whose nodes are dimensions in $\mathcal{D}$.* □

A *path* in a dimension graph is a sequence $D_1, \ldots, D_k$ of distinct nodes such that there is a directed edge from D_i to D_{i+1}, where $1 \leq i \leq k-1$.

Based on the definitions of dimension graphs as general structures, we proceed to define formally dimension graphs with respect to a value tree.

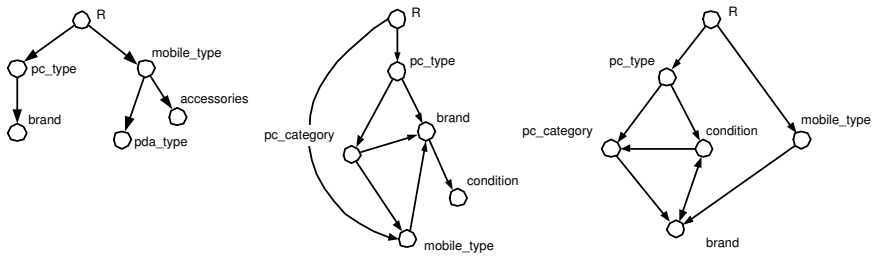
Definition 5. *Let T be a value tree over a dimension set $\mathcal{D}$. A* dimension graph *of T is a dimension graph (N, E), where N is a set of nodes and E is a set of edges defined as follows:*

(a) There is a node D in N iff 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 iff 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 *$\mathcal{G}$.* □

Example 4. Consider for example the value trees T_1, T_2 and T_3 of Figure 1. Figure 3 shows the dimension graphs $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ of T_1, T_2 and T_3, respectively. There is an edge from dimension `mobile_type` to dimension `pda_type` in $\mathcal{G}_1$, since a node labeled `Palm` (a value of `pda_type`) is a child of a node labeled `PDAs` (a



Dimension graph G_1 **(a)** Dimension graph G_2 **(b)** Dimension graph G_3 **(c)**

Fig. 3. Dimension Graphs

value of `mobile_type`) in value tree T_1. Looking at the lower left part of value tree T_3, we note that a node labeled `Mac` (a value of `brand`) is a child of a node labeled `New` (a value of `condition`). However, looking at the lower right part of T_3, a node labeled `New` is a child of a node labeled `Sony` (another value of dimension `brand`). Thus, there is an edge from dimension `condition` to dimension `brand` and an edge from `brand` to `condition` in $\mathcal{G}_3$ which are compactly shown in the figures by a double headed edge. □

The dimension graph of a value tree has a particular form. The following propositions describe some of its properties. Their proof is straightforward.

Proposition 1. There is exactly one node in the dimension graph of a value tree having only outgoing edges. □

This unique node is called *root* of the dimension graph. Dimension graphs can have cycles. For instance, in Figure 3(c), dimension graph $\mathcal{G}_3$ has two cycles: `pc_category, brand, condition, pc_category` and `condition, brand, condition`.

Proposition 2. For every node of a dimension graph, there is a path from the root to that node. □

The next proposition relates paths from the root in a value tree to paths from the root in its dimension graph.

Proposition 3. Consider a dimension graph $\mathcal{G}$ of a value tree T over a dimension set $\mathcal{D}$. Let $v_1, \ldots, v_k$ be values from the distinct dimensions $D_1, \ldots, D_k \in \mathcal{D}$, respectively. If $v_1, \ldots, v_k$ label, in that order, nodes on a path in T, then $D_1, \ldots, D_k$ appear in that order on a path from the root in $\mathcal{G}$. □

4 Queries

We present in this section a simple query language and we outline how queries can be evaluated. Our intension is not to provide a full-fledged language. For instance,

it does not include selection predicates. Our goal is to show how dimensions can be used to query value trees. Queries in this language are defined on dimension graphs. 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 interesting 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. The system can identify possible orderings of dimensions in the paths of the answer based on the dimension graph only. These orderings are used as patterns for constructing the path expressions that compute the answer of the query on the underlying value tree. All the other orderings of dimensions are excluded from consideration before the computation of the query answer reaches the value tree.

4.1 Syntax

A query on a dimension graph comprises annotations of the graph dimensions with sets of values and specifications of precedence relationships between the graph dimensions.

Definition 6. *Let $\mathcal{G}$ be a dimension graph over a dimension set $\mathcal{D}$. A* query *Q on $\mathcal{G}$ 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 the* annotation *of D_i in Q. 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. □

We graphically represent a query $Q = (\mathcal{A}, \mathcal{P})$ on a dimension graph $\mathcal{G}$ 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. A single (double) arrow $D_i \rightarrow D_j$ ($D_i \Rightarrow D_j$) denotes that the values used to annotate D_j should be children (descendants) of the values used to annotate D_i. The unqualified word "arrow" refers indiscreetly to a single or double arrow.

Example 5. Consider the dimension graphs $\mathcal{G}_1, \mathcal{G}_2$, and $\mathcal{G}_3$ of Figure 3. Figure 4 shows the graphical representation of different queries on these dimension graphs. Annotated nodes are shown in the figures with black circles. Precedence relationships are shown with single or double arrows from one node to another.

Figure 4(a) represents query $Q_1 = (\mathcal{A}_1, \mathcal{P}_1)$ on dimension graph $\mathcal{G}_1$, where $\mathcal{A}_1 = \{\texttt{brand} = \{\texttt{Mac}, \texttt{Sony}\}, \texttt{pc\_type} = \{\texttt{Desktop}\}\}$ and $\mathcal{P}_1 = \emptyset$. In Q_1 we do not specify any precedence relationships between the annotated notes.

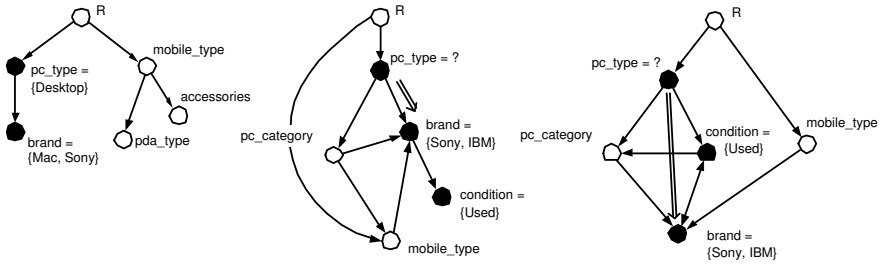


Fig. 4. Graphical Representation of Queries

Figure 4(b) represents query $Q_2 = (\mathcal{A}_2, \mathcal{P}_2)$ on dimension graph $\mathcal{G}_2$, where $\mathcal{A}_2 = \{\texttt{pc\_type} =?, \texttt{brand} = \{\texttt{Sony}, \texttt{IBM}\}, \texttt{condition} = \{\texttt{Used}\}\}$ and $\mathcal{P}_2 = \{\texttt{pc\_type} \Rightarrow \texttt{brand}\}$. A double arrow from node `pc_type` to node `brand` denotes the precedence relationship in $\mathcal{P}_2$.

Figure 4(c) represents query $Q_3 = (\mathcal{A}_3, \mathcal{P}_3)$ on dimension graph $\mathcal{G}_3$, where $\mathcal{A}_3 = \{\texttt{pc\_type} =?, \texttt{brand} = \{\texttt{Sony}, \texttt{IBM}\}, \texttt{condition} = \{\texttt{Used}\}\}$ and $\mathcal{P}_3 = \{\texttt{pc\_type} \Rightarrow \texttt{brand}\}$. Query Q_3 is identical to Q_2 but it is defined on dimension graph $\mathcal{G}_3$. □

In the following we often identify a query with its graphical representation.

4.2 Semantics

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 7. *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[3] 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 naming a node 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.

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

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

Clearly, the answer of a query that does not involve any annotations or arrows (this is the query $(\emptyset, \emptyset)$) on a value tree T is T itself. 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 value of the corresponding dimension in every root-to-leaf path in the answer of the query.

Example 6. Consider the queries Q_1, Q_2 and Q_3 on the dimension graphs $\mathcal{G}_1, \mathcal{G}_2$, and $\mathcal{G}_3$, respectively, graphically shown in Figure 4. Consider also the value trees T_1, T_2 and T_3 of Figure 1. Figure 5 shows the answers T_1', T_2' and T_3' of Q_1, Q_2 and Q_3 on T_1, T_2 and T_3, respectively.

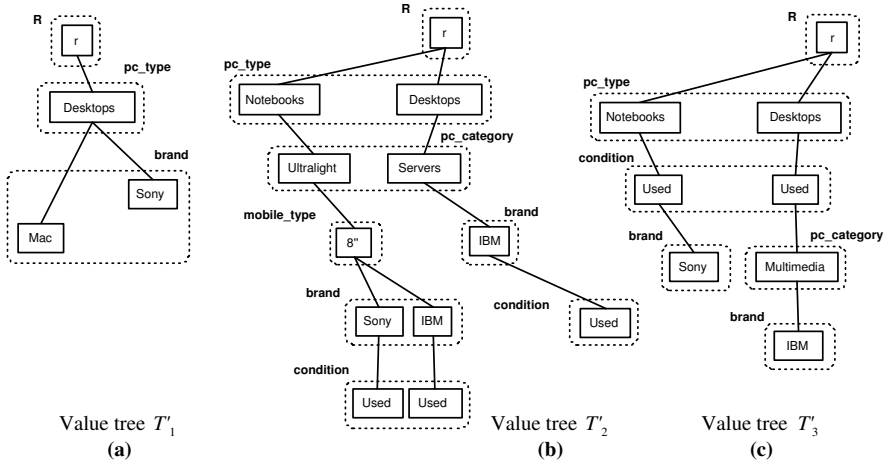


Fig. 5. Query Answers

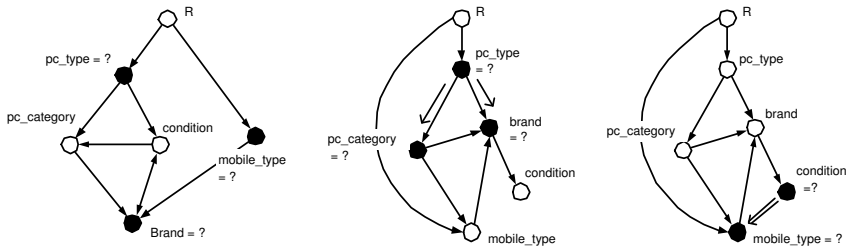
Further, consider the query $Q_4 = (\mathcal{A}_4, \mathcal{P}_4)$, where $\mathcal{A}_4 = \{$ `pc_type = {Desktops}, brand = {HP,Gateway}`$\}$, and $\mathcal{P}_3 = \{$`pc_type` $\rightarrow$ `brand`$\}$ on the dimension graph $\mathcal{G}_1$ shown in Figure 3. In the value tree T_1 shown in Figure 1(a) there are values of dimension `brand` that are children of values of dimension `pc_type`. However, there is no root-to-leaf path that involves values `Desktops` and `HP`, or `Desktops` and `Gateway`. Therefore, the answer of Q_4 on T_1 is empty. □

4.3 Unsatisfiable Queries

A query on a dimension graph $\mathcal{G}$ is called *unsatisfiable* if its answer is empty on every value tree underlying $\mathcal{G}$. Otherwise, it is called *satisfiable*. Detecting the unsatisfiability of a query saves its evaluation on a value tree (which, in any

case, produces an empty answer.) In general, this value tree is much larger than its dimension graph which might be needed for detecting the unsatisfiability of the query. The graphical representation of a query provides some intuition on unsatisfiable queries.

Example 7. Consider the dimension graphs $\mathcal{G}_2$ and $\mathcal{G}_3$ of Figure 3, and the queries Q_5 on $\mathcal{G}_3$, and Q_6 and Q_7 on $\mathcal{G}_2$ graphically represented in Figure 6. These queries are unsatisfiable.



Query Q_5 on G_3 **(a)** Query Q_6 on G_2 **(b)** Query Q_7 on G_2 **(c)**

Fig. 6. Unsatisfiable Queries

In query Q_5 of Figure 6(a), there is no path from the root of $\mathcal{G}_3$ that involves all the annotated nodes. By Proposition 3 there is no root-to-leaf path in a value tree underlying $\mathcal{G}_3$ that involves values for the annotated dimensions in Q_5.

In query Q_6 of Figure 6(b), there is a path from the root of $\mathcal{G}_2$ through all the annotated nodes (e.g. the path (R, `pc_type, pc_category, brand`)). However, there are two outgoing single arrows from the same node (node `pc_type`). Clearly, no two values can be children of the same node in a root-to-leaf path of a value tree underlying $\mathcal{G}_2$.

In query Q_7 of Figure 6(c), there is also a path from the root of $\mathcal{G}_2$ through all the annotated nodes (e.g. the path (R, `mobile_type, brand, condition`)). However, there is a double arrow from node `condition` to node `mobile_type` in Q_7 and no path from `condition` to `mobile_type` in $\mathcal{G}_2$. By Proposition 3 there is no root-to-leaf path in a value tree underlying $\mathcal{G}_2$ that involves a value for dimension `condition` preceding a value for dimension `mobile_type`. □

More generally, we can show the following result that provides sufficient conditions for a query to be unsatisfiable.

Proposition 4. A query Q on a dimension graph $\mathcal{G}$ is *unsatisfiable* if one of the following conditions holds:

(a) Arrows 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 ($D_i \neq D_j$).

(c) There is a precedence relationship $D_i \rightarrow D_j$ in Q but no edge from node D_i to node D_j in $\mathcal{G}$.
(d) There is a precedence relationship $D_i \Rightarrow D_j$ in Q but no edge from node D_i to node D_j in the *transitive closure* of $\mathcal{G}$ (in other words, no path from node D_i to node D_j in $\mathcal{G}$).
(e) The annotated nodes in Q are not on a path from the root of $\mathcal{G}$. □

Proof: Conditions (a), (b), (c) and (d) violate condition (e) of Definition 7, as follows:
(a) If there are arrows in Q that form a directed cycle, say $D_i \rightarrow D_j$ and $D_j \rightarrow D_i$, then a value of D_j should be the parent and the child of a value of D_i in the same path of the value tree, which is not possible since there could not be two nodes on a path in the value tree labeled by values that belong to the same dimension.
(b) If there are precedence relationships $D \rightarrow D_i$ and $D \rightarrow D_j$ in Q, then a value of D should be the parent of a value of D_i and a value of D_j in the same path of the value tree, which is not possible.
(c) If there is a precedence relationship $D_i \rightarrow D_j$ in Q but no edge from node D_i to node D_j in $\mathcal{G}$, then there is not any path with a value from D_i being the parent of a value from D_j in the value tree.
(d) If there is a precedence relationship $D_i \Rightarrow D_j$ in Q but no path from node D_i to node D_j in $\mathcal{G}$, then there is not any path with a value from D_i being the ancestor of a value from D_j in the value tree.
Condition (e) violates condition (a) of Definition 7: If the annotated nodes in Q are not on a path from the root of $\mathcal{G}$, then the resulting tree will not include value r as its root, and thus T' and T will not have the same root. □

In order to provide necessary conditions for query unsatisfiability, we introduce the concept of an answer path of a query.

Definition 8. *Let Q be a query on a dimension graph $\mathcal{G}$. An* answer path *of Q in $\mathcal{G}$ 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. □

Example 8. Consider the query Q_2 on dimension graph $\mathcal{G}_2$ and the query Q_3 on dimension graph $\mathcal{G}_3$, which are shown in Figures 4(b) and 4(c), respectively. One can identify the following answer paths for query Q_2 in $\mathcal{G}_2$:

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

The answer paths for query Q_3 in $\mathcal{G}_3$ are:

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

The following proposition provides necessary and sufficient conditions for a query to be unsatisfiable.

Proposition 5. A query Q on a dimension graph $\mathcal{G}$ is *unsatisfiable* iff there is no answer path of Q in $\mathcal{G}$. □

Proof: (If part) We show that if Q is satisfiable, there is an answer path of Q in $\mathcal{G}$. Assume that Q is satisfiable and let value tree T' be the answer of Q on a value tree T. Let $p' = r, v_1, \ldots, v_m$ be a root-to-leaf path in T'. By definition, p' is also a root-to-leaf path in T. Since p' satisfies the conditions of Definition 7, the path $p = R, D_1, \ldots, D_m$, where $v_i \in D_i,\ i = 1, \ldots, m$, satisfies the conditions of Definition 8. Therefore, p is an answer path of $\mathcal{G}$ in Q.

(Only if part) We show that if there is an answer path of Q in $\mathcal{G}$, Q is satisfiable. Let p be an answer path of Q in $\mathcal{G}$. Let T be the tree induced by a depth first traversal of $\mathcal{G}$ where every occurrence of a node D_i of $\mathcal{G}$ in T is labeled by a (the same) value v_i of dimension D_i. In particular, if D_i is a dimension in Q annotated by the set $\mathcal{A}_i$, $v_i \in \mathcal{A}_i$. If $p = R, D_1, \ldots, D_k$, there is a root-to-leaf path $p' = r, v_1, \ldots, v_m,\ k \leq m$, in T such that $v_i \in D_i,\ i = 1, \ldots, k$. Since p satisfies the conditions of Definition 8, p' satisfies the conditions of Definition 7. Therefore, p' is a root-to-leaf path in the answer of Q on T. Consequently, the answer of Q in T is non-empty. We conclude that Q is satisfiable. □

4.4 Query evaluation

When evaluating a query, we first check it for satisfiability. If a query is satisfiable, we proceed to compute its answer on a value tree in three steps. In the first step, we compute all the answer paths of the query. In the second step, we generate path expressions based on the answer paths. In the third step we evaluate the path expressions on the value tree and compose the answer of the query.

To represent path expressions, we use a notation similar to that of XPath [4]. The fragment of XPath we use involves node names (v_i), child axis ($/$), descendant axis ($//$), wildcards ($*$), unions ($|$). The expression $(v_1|\ldots|v_m)$ represents any node name in the set $\{v_1, \ldots, v_m\}$. For a dimension D, we use the expression $*_D$ as an abbreviation for the expression $(v_1|\ldots|v_n)$, where $\{v_1, \ldots, v_n\} = D$.

Given an answer path, we construct a corresponding path expression as follows. Let $R, D_1, \ldots, D_k$ be an answer path of a query Q. The corresponding path expression has the form $r/\theta_1/\ldots/\theta_k$, where, for $i = 1, \ldots, k$,

$$\theta_i = \begin{cases} (v_1|\ldots|v_m) & \text{if } D_i \text{ is annotated with the value set } \{v_1, \ldots, v_m\} \\ *_{D_i} & \text{if } D_i \text{ is annotated with a “?” or if } D_i \text{ is not annotated} \end{cases}$$

Notice that even though nodes annotated with a “?” are treated the same way as non-annotated ones in the construction of path expressions for a query, they affect differently the answer of a query since they are taken into account in the identification of answer paths for that query.

Before showing what the result of a path expression on a value tree is, we introduce the concept of a merge of a set of value trees (or paths). Let $T_1, \ldots, T_k$ be a set of value trees having the same root r. The *merge* of $T_1, \ldots, T_k$, denoted $T_1 \cup \ldots \cup T_k$, is a minimal[4] value tree which has $T_1, \ldots, T_k$ as subtrees. It is not difficult to see that this value tree is unique.

We show now what is the result of a path expression on a value tree. Let e be a path expression and T be a value tree. Let also P be the set of paths from the root of T to the leafs of T that satisfy e. The result $res(e, T)$ of a path expression e on a value tree T is the value tree $\bigcup_{p \in P} p$. Note that the result of a path expression is different than the result of the same XPath expression. The result of a path expression is a value tree while the result of the same XPath expression is a set of nodes [4]. We can use XQuery [5] to compute the result of a path expression as it is defined here.

The answer of a query on a value tree can be computed by merging the results of its path expressions on the value tree. Let $E = \{e_1, \ldots, e_n\}$ be the set of path expressions constructed from all the answer paths of a query Q. The answer of Q on a value tree T is the value tree $\bigcup_{i \in [1,n]} res(e_i, T)$.

Example 9. Consider the query Q_2 on dimension graph $\mathcal{G}_2$, which is shown in Figure 4(b). The answer paths for Q_2 in $\mathcal{G}_2$ are shown in Example 8. These answer paths generate the following path expressions:

$r/ *_{\texttt{pc\_type}} /(\texttt{Sony}|\texttt{IBM})/\texttt{Used}$
$r/ *_{\texttt{pc\_type}} / *_{\texttt{pc\_category}} /(\texttt{Sony}|\texttt{IBM})/\texttt{Used}$
$r/ *_{\texttt{pc\_type}} / *_{\texttt{pc\_category}} / *_{\texttt{mobile\_type}} /(\texttt{Sony}|\texttt{IBM})/\texttt{Used}$

Evaluating these path expressions on the value tree T_2 of Figure 1(b), one can see that the result of the first path expression is an empty value tree. In contrast, the second path expression contributes one path, while the third one contributes two paths to the answer of Q_2 on T_2 (Figure 5(b)).

Consider also the query Q_3 on dimension graph $\mathcal{G}_3$, which is shown in Figure 4(c). The answer paths for Q_3 in $\mathcal{G}_3$ are shown in Example 8. They generate the following path expressions:

$r/ *_{\texttt{pc\_type}} /\texttt{Used}/(\texttt{Sony}|\texttt{IBM})$
$r/ *_{\texttt{pc\_type}} /\texttt{Used}/ *_{\texttt{pc\_category}} /(\texttt{Sony}|\texttt{IBM})$
$r/ *_{\texttt{pc\_type}} / *_{\texttt{pc\_category}} /(\texttt{Sony}|\texttt{IBM})/\texttt{Used}$

Of those path expressions, evaluating the third one on the value tree T_3 of Figure 1(c) results in an empty value tree. Only the first two contribute paths to the answer of Q_3 on T_3 (Figure 5(c)). □

5 Data Source Integration

There are two major approaches to integrating data sources: the materialized (or data warehousing) and the virtual (or mediated) [32]. In the materialized

[4] Minimality is meant with respect to the number of nodes or edges.

approach, data are extracted from the data sources, integrated, and stored in a repository (the data warehouse). User queries are addressed to the data warehouse and are evaluated locally. When the data sources change, the stored data need to be maintained. This is a drawback of the materialized approach for applications that require current data. In the virtual approach, a global (or mediated) logical schema is used purely for the purpose of user query formulation [20]. Mappings are established between the global schema and the schemas of the data sources. User queries are posed on the global schema and they are transformed to queries on the data sources using these mappings. These later queries are sent to and evaluated at the data sources and their answers are sent back and combined for presentation to the user. The virtual approach guarantees that answer data are always current. Hybrid approaches combine features of the materialized and virtual approaches [21].

We follow a virtual approach adapted to the type of data structures we consider here. There are two key differences with respect to a traditional virtual approach:

(a) Since we are dealing with tree structures we do not have schemas. User queries are posed on a dimension graph constructed by integrating the dimension graphs of the value trees of the different data sources. We qualify this new dimension graph as *global* as opposed to the dimension graphs of the data sources which are characterized as *local*.

(b) Queries on the global dimension graph (*global queries*) are translated into queries on the dimension graphs of the data sources (*local queries*). However, a global query does not need any transformation to become a local query. It can either be defined on a local dimension graph or not.

Figure 7 shows the architecture of our data integration system. Each data source i contains a value tree T_i and its local dimension graph $\mathcal{G}_i$. Each value in T_i is mapped to a dimension. The global site contains the global dimension

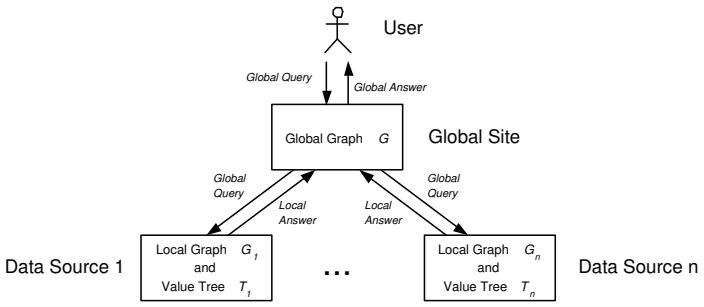


Fig. 7. A data integration system architecture

graph. We show how a global dimension graph is constructed in Section 5.1. Global queries are posed by the users on this dimension graph. They are formally defined in Section 5.2. Global queries are checked for satisfiability at the global

site. They are potentially forwarded to the data sources where they are evaluated on the local dimension graphs and value trees, and their answers are sent back to the global site. The process of evaluating global queries is described in Section 5.3.

5.1 Global Dimension Graph Construction

The global dimension graph is constructed from the local dimension graphs.

Definition 9. *Let $\mathcal{G}_1 = (N_1, E_1), \ldots, \mathcal{G}_n = (N_n, E_n)$ be local dimension graphs of value trees over dimension set $\mathcal{D}$. A* global dimension graph $\mathcal{G}$ *for $\mathcal{G}_1, \ldots, \mathcal{G}_n$ is a dimension graph (N, E), where N is a set of nodes and E is a set of edges defined as follows:*

(a) $N = N_1 \cup \ldots \cup N_n$.
(b) There is a directed edge in E from node D_i to node D_j iff there is a directed edge from node D_i to node D_j in some local dimension graph. □

Example 10. Consider the local dimension graphs $\mathcal{G}_1, \mathcal{G}_2$, and $\mathcal{G}_3$ of Figure 3. Figure 8 shows the global dimension graph $\mathcal{G}$ for $\mathcal{G}_1, \mathcal{G}_2$, and $\mathcal{G}_3$.

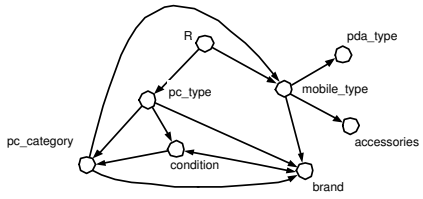


Fig. 8. The global dimension graph $\mathcal{G}$ for $\mathcal{G}_1, \mathcal{G}_2$ and $\mathcal{G}_3$

Notice that $\mathcal{G}$ has cycles (e.g. `pc_category`,`mobile_type`, `brand`, `condition`, `pc_category`) that do not appear in any of the local dimension graphs. □

5.2 Queries on Global Dimension Graphs

The syntax of global queries is identical to that of local queries (see Section 4.1). Global queries are posed on global graphs.

Example 11. Consider the global dimension graph $\mathcal{G}$ of Figure 8. Figure 9(a) shows a graphical representation of query Q_8 on $\mathcal{G}$. As with the representation of local queries, annotated nodes are shown with filled black circles. □

The answer of a global query is defined in terms of the answer of a local query.

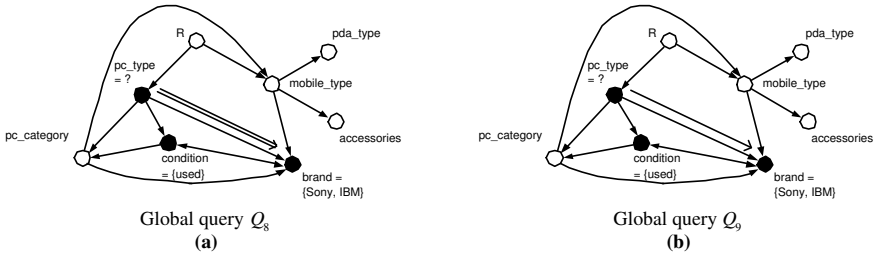


Fig. 9. Global queries Q_8 and Q_9

Definition 10. *Let $\mathcal{G}_1, \ldots, \mathcal{G}_n$ be local dimension graphs of value trees $T_1, \ldots, T_n$ respectively over dimension set $\mathcal{D}$. Let also $\mathcal{G}$ be a global dimension graph for $\mathcal{G}_1, \ldots, \mathcal{G}_n$ and Q be a query on $\mathcal{G}$. The* answer *of Q on the list of value trees $T_1, \ldots, T_n$ is a list $T'_1, \ldots, T'_n$, where each T'_i, $i = 1, \ldots, n$, is defined as follows:*

1. *If there is an annotated dimension in Q that does not appear in $\mathcal{G}_i$, then T'_i is ϵ (empty answer). In this case, we say that the global query Q is not* definable *on the local dimension graph $\mathcal{G}_i$.*
2. *Otherwise, let Q_i be the query Q on the local dimension graph $\mathcal{G}_i$. T'_i is the answer of query Q_i on T_i.* □

Example 12. Consider the global query Q_8 of Figure 9(a) and the local value trees T_1, T_2 and T_3 of Figure 1. The local dimension graphs $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ of the local value trees T_1, T_2 and T_3 are shown in Figure 3. Query Q_8 annotates dimension `condition` that does not appear in $\mathcal{G}_1$. All the annotated dimensions of Q_8 appear in $\mathcal{G}_2$ and $\mathcal{G}_3$. Query Q_8 on $\mathcal{G}_2$ is the local query Q_2 shown in Figure 4(b), while query Q_8 on $\mathcal{G}_3$ is the local query Q_3 shown in Figure 4(c). The answers of Q_2 and Q_3 on the local trees T_2 and T_3 underlying the local dimension graphs $\mathcal{G}_2$ and $\mathcal{G}_3$ are the value trees T'_2 and T'_3 shown in Figure 5(b) and 5(c) respectively. Therefore, the answer of Q on T_1, T_2, T_3 is ϵ, T'_2, T'_3. □

A (global) query on a global dimension graph $\mathcal{G}$ is called *unsatisfiable* if its answer is a list of empty answers on every list of value trees that have $\mathcal{G}$ as the global dimension graph of their local dimension graphs. Otherwise, it is called *satisfiable*. Clearly, the necessary and sufficient conditions provided by Proposition 5 for a local query to be unsatisfiable also hold for a global query.

5.3 Global Query Evaluation

Global query evaluation can be described in three phases. The first phase involves the global dimension graph, the second phase the local dimension graphs and the third phase the value trees.

In the first phase, the global query is checked for satisfiability at the global site. This check involves only the global query and the global dimension graph. If the query is satisfiable, it is sent to the data sources.

In the second phase, each data source checks if the global query is definable on its local dimension graph. If a global query is not definable at a data source, an empty answer is returned to the global site. Otherwise, it is checked for satisfiability as a local query. Notice that a query on a local graph may be unsatisfiable even if the same query on the global graph is satisfiable. The reason is that a local graph may be more restricted than the global graph: there may be paths or directed edges between two nodes in the global graph that do not exist between the same nodes on a local graph. Conditions (c), (d), and (e) of Proposition 4 show that the lack of these paths or edges may imply the unsatisfiability of the local query. If a local query is unsatisfiable, an empty answer is sent to the global site by the corresponding data source.

Example 13. Consider the global query Q_9 on the global dimension graph $\mathcal{G}$ graphically represented in Figure 9(b). This query involves an arrow from node `pc_type` to node `brand`. It is satisfiable since there is an edge in $\mathcal{G}$ from node `pc_type` to node `brand`. In contrast, the same query on the local dimension graph $\mathcal{G}_3$ of Figure 4(c) is unsatisfiable since there is no such an edge $\mathcal{G}_3$. □

In the third phase, satisfiable local queries are evaluated as described in Section 4, and the answers are sent to the global site for presentation to the user. Figure 10 outlines the different phases of the evaluation of a global query.

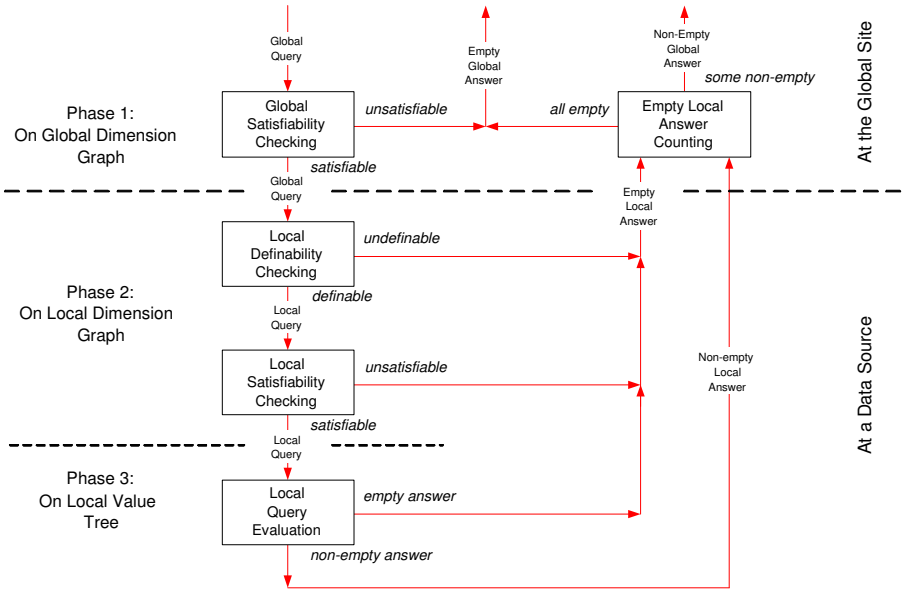


Fig. 10. Global Query Evaluation

6 Experimental Evaluation

We implemented and compared our integration strategy against one that does not exploit dimension graphs to query multiple tree-structured data sources. We briefly summarize the two integration strategies:

$A1$: Queries are formed on the global dimension graph. The evaluation of a query on a value tree is performed only if the query is satisfiable on the global dimension graph, as well as definable and satisfiable on the local dimension graph of this value tree. 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. This approach does not exploit dimension graphs for the evaluation of queries.

In order to maintain similar query sets for both approaches, our experimental platform transforms queries on the global dimension graph that involve dimensions into sets 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)`.

We used a set of synthetic value trees, and we measured the execution time for evaluating queries on a global dimension graph. The set of value trees was constructed as follows. We generated a random set of 30 dimensions with 10 values each (a total of 300 distinct values), and we created 10 random value trees using those values. The actual number of values in the value trees ranges from 280 to 500. Queries were generated by randomly annotating dimensions in dimension graphs and adding arrows. In order to add arrows to the annotated dimensions 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.

6.1 Experiments and Results

We carried out three different types of experiments[5] to study the differences in the execution time of the two integration approaches. For every measure point in the x-axis, 10 queries were generated for each one of the 10 value trees. The recorded execution time per point is the average execution time.

[5] All the experiments were carried out on an AMD Sempron 2600 PC with 512MB RAM.

Varying the size of the queries.

Satisfiable and unsatisfiable 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 11, we present the results obtained for global queries having 2 to 8 annotated dimensions, varying the percentage of arrows. The y-axis is on a logarithmic scale. The number of dimensions is fixed to 30. In each query, 50% of the arrows were single (parent/child relationships) and 50% were double (ancestor/descendant relationships). In this particular experiment, 50% of queries executed were unsatisfiable. In any case, the approach $A1$ clearly outperforms $A2$.

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.

Only satisfiable queries. We performed the previous experiment with only satisfiable queries. We observed that approach $A1$ outperforms approach $A2$ even in this case. Figure 12 shows, for example, the results obtained for queries having 3 and 4 annotated dimensions, varying the percentage of arrows.

Varying the type of arrows in the queries.

Satisfiable and unsatisfiable queries. We measured the execution time varying the percentage of single arrows in the total number of arrows in the global query

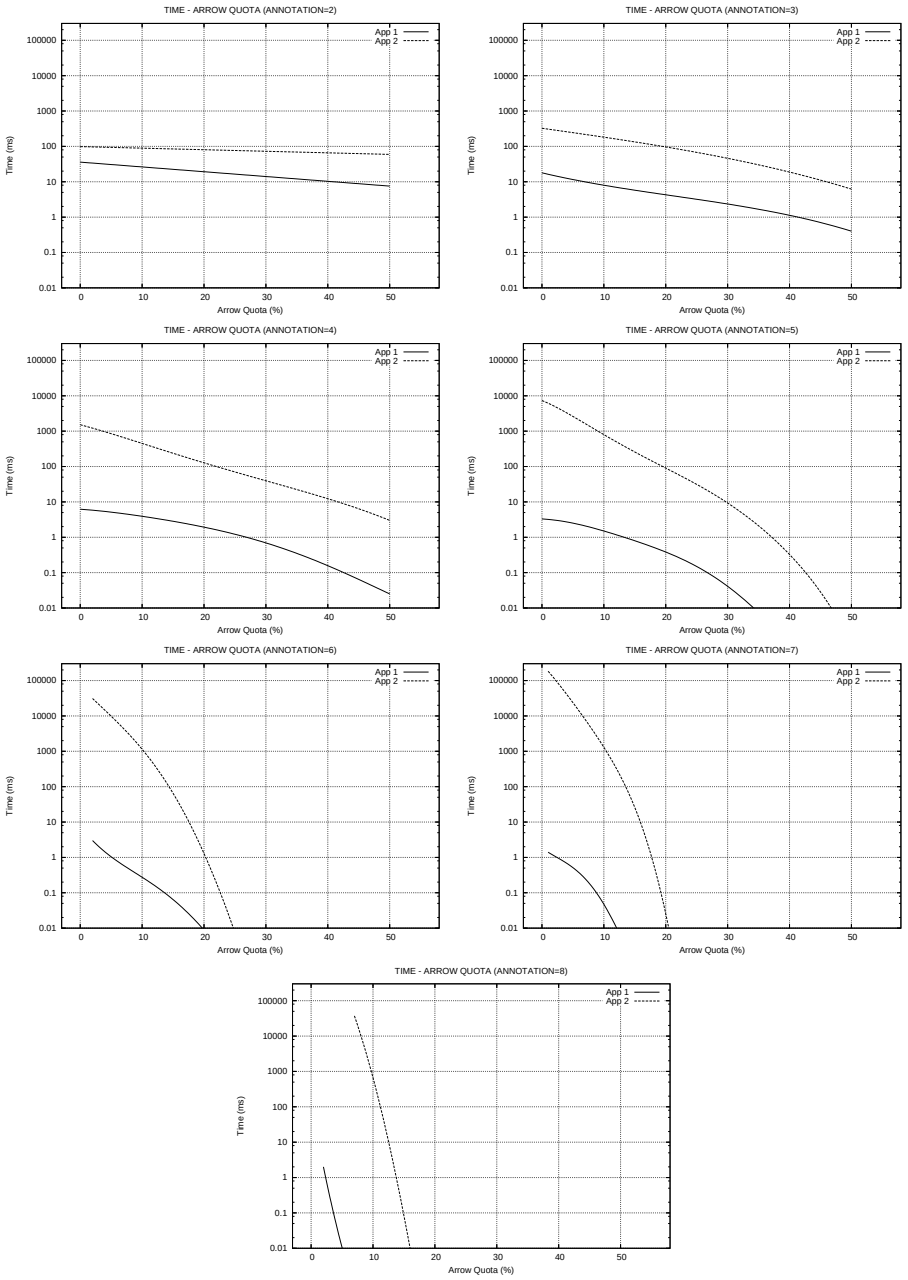


Fig. 11. Execution time varying the percentage of arrows for different numbers of annotated dimensions in the global query (50% of queries executed were unsatisfiable).

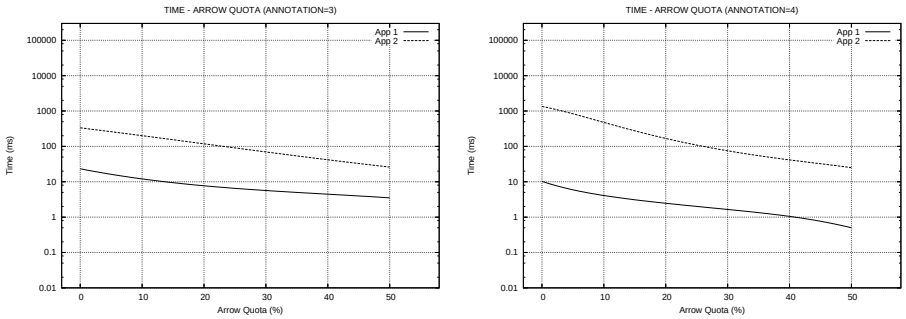


Fig. 12. Execution time varying the percentage of arrows for different numbers of annotated dimensions in the global query (all queries executed were satisfiable).

for different pairs of numbers of annotated dimensions and arrows. In Figure 13, 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 less answers paths. The reason is that the 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 global dimension graph and the local dimension graphs to detect unsatisfiable queries, and to reduce the number of path expressions generated. The approach $A1$ outperforms $A2$ by three to four orders of magnitude.

Only satisfiable queries. We performed the previous experiment with only satisfiable queries. Even in this case approach $A1$ outperforms approach $A2$. Figure 14 shows for example the results obtained for queries having (a) 6 annotated dimensions and 4 arrows, and (b) 7 annotated dimensions and 5 arrows.

Varying the size of dimension graphs.

Satisfiable and unsatisfiable queries. We calculated the execution time for different sizes of global dimension graphs, varying the percentage of arrows. The number of annotated dimensions is fixed for different numbers of dimensions. In Figure 15, we present the results obtained for global dimension graphs having 30, 34, 38 and 42 dimensions. In each global 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 global dimension graph. Starting from a fixed value tree, its partition and a set of

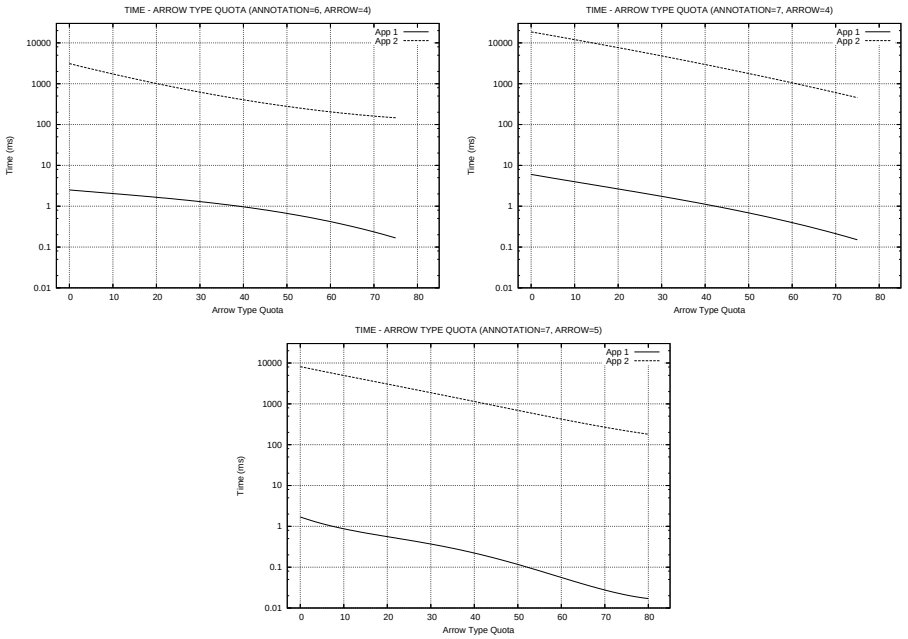


Fig. 13. Execution time varying the percentage of single arrows in the total number of arrows in the global query, for different pairs of numbers of annotated dimensions and arrows (50% of queries executed were unsatisfiable).

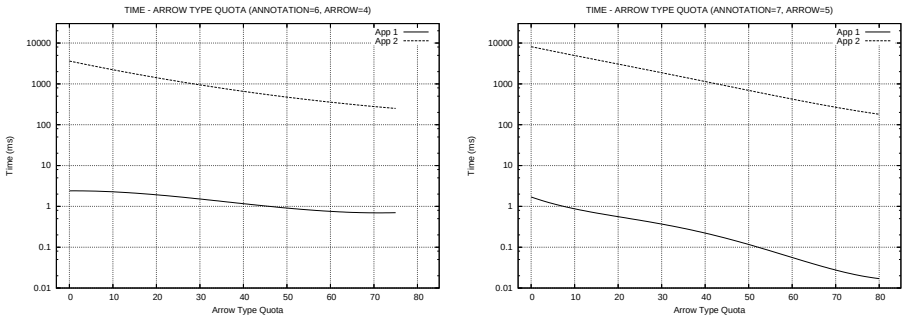


Fig. 14. Execution time varying the percentage of single arrows in the total number of arrows in the global query, for different pairs of numbers of annotated dimensions and arrows (all queries executed were satisfiable).

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 the global dimension graph. The approach $A1$ clearly outperforms $A2$.

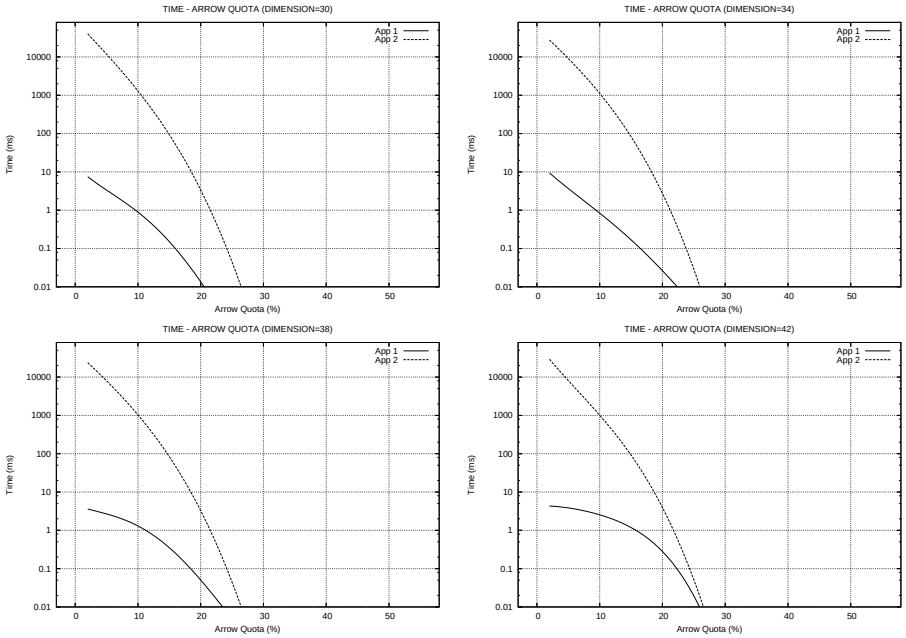


Fig. 15. Execution time varying the percentage of arrows for different numbers of dimensions in the global dimension graph (50% of queries executed were unsatisfiable).

Only satisfiable queries. Approach $A1$ outperforms approach $A2$ even when the previous experiment is ran with only satisfiable queries. Figure 16 shows, for example, the results obtained for 34 dimensions.

7 Conclusions

We presented a method for integrating value trees that have structural differences and inconsistencies. Our approach exploits semantic information for the nodes of value trees. A semantic relationship between nodes in value trees was captured by the concept of a dimension. Dimension graphs were defined to capture structural information on the dimensions of a value tree. However, dimension graphs are

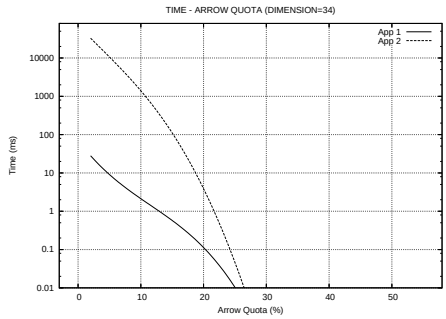


Fig. 16. Execution time for 34 dimensions in the global dimension graph (all queries executed were satisfiable).

not plain structural summaries of value trees, but rather semantically richer constructs that are able to support semantic integration of tree-structured data. We designed a query language to query value trees. Queries are specified on the dimensions of the value tree and can optionally involve parent-child and ancestor-descendant relationships between these dimensions. We provided necessary and sufficient conditions for query unsatisfiability and we presented a technique for evaluating satisfiable queries. We defined global dimension graphs by merging dimension graphs of local data sources. Queries are issued on the dimensions of a global dimension graph. Thus, in our approach for integrating data sources, a query is not restricted by the structure of a specific local value tree. We presented a technique to evaluate queries issued on global dimension graphs, first on the global site and then on local data sources. We conducted experiments to compare our integration method against one that does not exploit dimension graphs to query multiple tree-structured data sources. Our results demonstrated the clear superiority of our approach.

Our future work will focus on the problem of determining dimensions for the nodes of value trees. This is called dimensioning problem. Currently we assume that the assignment of nodes to dimensions is given. We are interested in studying how this process can be supported by the use of classification hierarchies and ontologies and how such a semi-automatic method would affect our approach.

References

1. Exchangeable Faceted Metadata Language, (XFML), 2003, http://www.xfml.org/.
2. XML Topic Maps (XTM), 2001, http://www.topicmaps.org.
3. World Wide Web Consortium site (W3C), http://www.w3c.org.
4. XML Path Language (XPath). World Wide Web Consortium site (W3C), 2003-2005, http://www.w3c.org/TR/xpath20/.
5. XML Query (XQuery). World Wide Web Consortium site (W3C), The Architecture Domain. 2003-2005, http://www.w3.org/XML/Query.

6. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, California, 2000.
7. B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Ontology-based integration of XML web resources. In *Proceedings of the 1st International Semantic Web Conference (ISWC'02)*, Sardinia, Italy, June 2002.
8. S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *Proceedings of the 8th Conference on Extending Database Technology (EDBT'02)*, Prague, Czech Republic, Mar 2002.
9. R. Behrens. A grammar based model for XML schema integration. In *Proceedings of the 17th British National Conference on Databases (BNCOD'00)*, Exeter, UK, Jul 2000.
10. S. Bergamaschi, F. Guerra, and M. Vincini. A data integration framework for e-commerce product classification. In *Proceedings of the 1st International Semantic Web Conference (ISWC'02)*, Sardinia, Italy, Jun 2002.
11. P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceeding of the 6th International Conference on Database Theory (ICDT'97)*, Delphi, Greece, Jan 1997.
12. S. D. Camillo, C. A. Heuser, and R. dos Santos Mello. Querying heterogeneous XML sources through a conceptual schema. In *Proceeding of the 22nd International Conference on Conceptual Modeling (ER'03)*, Chicago, IL, USA, Oct 2003.
13. A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management*. Addison Wesley, 2003.
14. V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. In *Proceedings of the International Conference on Management of Data (ACM SIGMOD'00)*, Dallas, Texas, USA, May 2000.
15. S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale XML repository. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, Rome, Italy, Sep 2001.
16. R. dos Santos Mello and C. A. Heuser. A bottom-up approach for integration of XML sources. In *Proceedings of the International Workshop on Information Integration on the Web (WIIW'01)*, Rio de Janeiro, Brazil, Apr 2001.
17. M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *Proceedings of the International Conference on Management of Data (ACM SIGMOD'00)*, Dallas, Texas, USA, May 2000.
18. R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, Athens, Greece, Aug 1997.
19. S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In *Proceedings of the International Conference on Management of Data (ACM SIGMOD'02)*, Madison, USA, Jun 2002.
20. A. Halevy. Data integration: a status report. In *Proceedings of the Datenbanksysteme fur Business, Technologie und Web (BTW'03)*, 2003.
21. R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *Proceedings of the 16th Symposium on Principles of Database Systems (ACM PODS'97)*, Tucson, Arizona, May 1997.
22. D. Kim, J. Kim, and S.-G. Lee. Catalog integration for electronic commerce through category-hierarchy merging technique. In *Proceedings of the 12th International Workshop on Research Issues in Data Engineering (RIDE'02)*, San Jose, USA, Mar 2002.

23. M. L. Lee, L. H. Yang, W. Hsu, and X. Yang. XClust: Clustering XML schemas for effective integration. In *Proceedings of the 11th International Conference on Information and Knowledge Management (CIKM'02)*, McLean, Virginia, USA, Nov 2002.
24. M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the 21st Symposium on Principles of Database Systems (ACM PODS'02)*, Madison, Wisconsin, USA, Jun 2002.
25. I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneous data sources. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, Rome, Italy, Sep 2001.
26. P. J. Marron, G. Lausen, and M. Weber. Catalog integration made easy. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03) (poster)*, Bangalore, India, Mar 2003.
27. N. Polyzotis and M. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proceedings of the International Conference on Management of Data (ACM SIGMOD'02)*, Madison, USA, Jun 2002.
28. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334 – 350, 2001.
29. S. Ram and V. Ramesh. *Management of Heterogeneous and Autonomous Database Systems*. Morgan Kaufmann Publishers, 1999.
30. D. Theodoratos and T. Dalamagas. Querying tree-structured data using dimension graphs. In *Proceedings of 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, Porto, Portugal, Jun 2005.
31. Y. Tzitzikas, N. Spyratos, P. Constantopoulos, and A. Analyti. Extended faceted taxonomies for web catalogs. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE'02)*, Grand Hyatt, Singapore, Dec 2002.
32. J. Widom. Research problems in data warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM'02)*, Baltimore, Maryland, USA, Dec 1995.