Flexible XML Querying using Skyline Semantics

Sara Cohen, Maayan Shiloach

School of Computer Science and Engineering
The Hebrew University of Jerusalem

***Abstract*— Preferences over results of an XML query are of two distinct flavors. First, the user may prefer results which contain desired values, e.g., lower prices, favorite foods, higher ratings. Second, the user may prefer results with a certain structure, e.g., existence of a "discount" node, existence of an edge (and not only a path) between "departure" and "arrival" nodes. The first type of preference has been studied extensively over relational data, using skyline semantics, but has barely been considered for XML. The second type of preference has been studied for XML in the context of inexact querying, using scoring functions to rank results.**

This paper presents a query language for XML that incorporates both value-based and structural desires. Skyline semantics is used to determine optimal results. Algorithms for query evaluation under skyline semantics are presented and experimentation proves efficiency.

The paper is novel in three aspects. First, it considers *skyline querying over XML data values,* and not over values in a relational database. Second, it presents a method for *inexact querying of the structure of XML* that is based on computing a skyline, instead of using scoring functions. Third, it *combines* both types of user preference into a single language. These facets join together to yield a versatile language for flexible querying of XML.

I. Introduction

Traditional queries are of a rigid nature, in which users specify exactly their information need. Query results must exactly satisfy the query, and are all of equal quality, and hence, are not ranked. Recently, there has been a realization that users cannot always specify precisely their information needs, because of two reasons:

1) A user may have an idealized vision of the data he would like to find. This vision is based on his subjective preferences, e.g., for lower prices, favorite foods, higher ratings. However, a precise query expressing the user's greatest desires is likely to be unsatisfiable, especially since values for different attributes are often correlated, e.g., highly rated restaurants tend to have higher prices.
2) Over an XML database, the user may not be sufficiently familiar with the structure, in order to precisely specify a query. Even if the user is familiar with the schema of the document, the inherent flexibility of XML can allow many variations in the structure, which the user must choose between, in order to define an exact query. Finally, the structure of an XML document can itself be "data" over which the user has preferences, e.g., the user may prefer existence of a "discount" node, or existence of an edge (and not only a path) between "departure" and "arrival" nodes in a database of flights.

Skyline queries were introduced to deal with the first source of difficulty in specifying exact queries, over *relational databases*. In a skyline query, the user states his preferences, and in return receives all non-dominated answers. Efficient algorithms for skyline queries over relational database have been developed, e.g., [1]–[6]. However, skyline queries over XML have not been considered in the past.

Languages for *inexact queries* over XML have been introduced to deal with the second source of difficulty in specifying exact queries [7]–[12]. In these works, semantics for answering inexact queries over XML are presented. Usually, query answers are required to be *maximal* in some sense. In [10], [11] ranking of results was not considered. In [7]–[9], [12] ranking is *score-based*. In score-based ranking, the user provides a score for each query requirement, and the ranking of a result is computed by aggregating together scores of satisfied query components. Score-based ranking functions are problematic in that they determine a relative ordering of every pair of answers, even though, conceptually, some pairs are incomparable. We demonstrate this problem in an example.

Example 1.1: Database researcher Sam is planning on attending ICDE 2009 in China. He needs to find a flight from his home-town in the US to Shanghai. Sam is patriotic (and hence would like a US airline carrier), cheap (wants a discounted flight) and pressed for time (and wants a direct flight). Given these desires, a discounted flight with a US carrier, and a stop-over is clearly superior to a non-discounted flight with a US carrier, and a stop-over. However, neither of these flights can be directly compared to a discounted flight with a non-US carrier and no stop-over. Any score-based ranking method must compare such flights. □

This paper presents a query language for XML that incorporates both value-based and structural desires. Skyline semantics is used to determine optimal results. Queries with precise structure and preferences over data values, express the traditional notion of skyline queries, but over XML. Queries with preferences over the structure, and none over the data values, can be seen as traditional inexact queries, with the exception of using skyline semantics to determine optimal results, instead of a scoring function. Finally, by combining preferences on data and on structure, the user can go beyond previous languages and formulate extremely flexible queries over XML.

The contributions of this paper are as follows. First, a powerful language for inexact querying of XML data and structure is developed (Section II). Second, algorithms for efficiently evaluating preference queries are presented (Section III). Third, a method for determining satisfiability of

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.24

IEEE computer society

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.

queries over XML is presented (Section IV). This result helps to speed up query evaluation in our system and is also of independent value. Fourth, experimentation proves the efficiency and scalability of our algorithms (Section VI).

II. Framework

A. Databases

A *database* D is a directed, labeled *tree*. We use $N(D)$ and $E(D)$ to denote the nodes and edges, respectively, of D. We use $l(n)$ to denote the label of a node $n \in N(D)$. A node n can have textual content, denoted $tc(n)$.

Given $n, m \in N(D)$, we say that m is a *child* of n if $(n, m) \in E(D)$. We say that m is a *descendent of* n if $(n, m) \in E^*(D)$, where $E^*(D)$ is the transitive closure of $E(D)$.

Figure 1 contains two databases D_1 and D_2 that will be used as running examples in this paper. Note that node labels are written in black regular font, whereas node textual content is written in red italics. Both D_1 and D_2 contain information about hotels. In D_1, the data is grouped by country, i.e., underneath each country node is a set of hotels located in that country. In D_2, the data is grouped by hotel chain, i.e., underneath each hotel chain is the list of all hotels in that particular chain.

B. Exact Queries

An *exact query*, or simply a *query* for short, $Q = (V, E, C)$ is a rooted directed graph with labeled variables (i.e., nodes) V, edges E and constraints C. We use $l(v)$ to denote the label of variable $v \in V$. The edges in E may be of two types: child edges, written $\mathsf{child}(u, v)$, and descendent edges, written $\mathsf{desc}(u, v)$. The set C contains *constraints*. Each constraint c is either simply **true** or is of the form $v\,\theta\,s$ where $\theta \in \{=, \neq, \not\sim, \sim, \leq, \geq, <, >\}$ and s is a constant. We use $\sim$ ($\not\sim$) to denote (non-)containment of the value s in the textual content of a node. The rest of the operators are defined in the obvious manner.

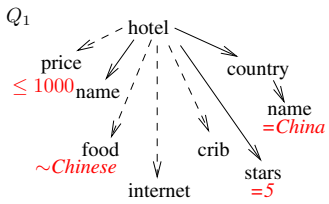


Fig. 2. Exact Query Q_1

Example 2.1: Sam needs a hotel to stay at when attending ICDE 2009 in China. Ideally, Sam would like a cheap (at most 1000RMB per night), 5-star hotel in China. Since he would like to taste the local cuisine, he would like Chinese food to be served at the hotel. Sam needs an Internet connection, to keep in touch. Finally, Sam will be bringing his wife and new baby, and so will need a crib.

Query Q_1 in Figure 2 expresses Sam's hotel desires precisely. Note that we use dotted lines for descendent edges and full lines for child edges. Constraints on textual content appear in red italics. □

We will use $V(Q)$ to denote the variables of Q, $E(Q)$ to denote the edges of Q and $C(Q)$ to denote the set of constraints in Q. If $E(Q)$ forms a tree, then we say that Q is a tree query. Tree queries, and more generally, graph-structured queries, are often used as a query language for XML databases.

Let D be a database and Q be a query. Let γ be a mapping of the variables in Q to the nodes in D. We say that γ *satisfies* an edge $e \in E(Q)$, written $\gamma \models e$, if

- $e = \mathsf{child}(u, v)$ and $(\gamma(u), \gamma(v)) \in E(D)$ *or*
- $e = \mathsf{desc}(u, v)$ and $(\gamma(u), \gamma(v)) \in E^*(D)$.

We say that γ satisfies a variable $v \in V(Q)$, if $l(\gamma(v)) = l(v)$. Finally, we say that γ satisfies a constraint $c = v\,\theta\,s \in C(Q)$, written $\gamma \models c$, if $tc(\gamma(v))\,\theta\,s$.

Given a database D and a query Q, the result of applying Q to D is the set $\mathsf{Ans}(Q, D)$ of mappings $\gamma : V(Q) \to N(D)$ that satisfy all variables, edges and constraints in Q. Observe that a cyclic query always returns an empty answer over all databases, since databases are *trees*. In acyclic queries, variables with multiple parents imply joins.

Example 2.2: Consider the result of evaluating Q_1 over D_1. Clearly $\mathsf{Ans}(Q_1, D_1) = \emptyset$, since Q_1 looks for a node labeled hotel that is the parent of a node labeled country. However, the node hierarchy in D_1 does not follow this ordering. When evaluating Q_1 over D_2, the result is once again empty. To see this, note, for example, that the China hotel in D_2 does not have a low enough price. □

C. Preference Queries

Exact queries impose very precise conditions on the satisfying mappings. As discussed earlier, when querying XML, the user may not always be able to provide such precise requirements. For such scenarios, we define the notion of *preference queries*, which allow the user flexibility in defining his requirements.

Intuitively, a preference query is similar to an exact query, with three changes/additions:

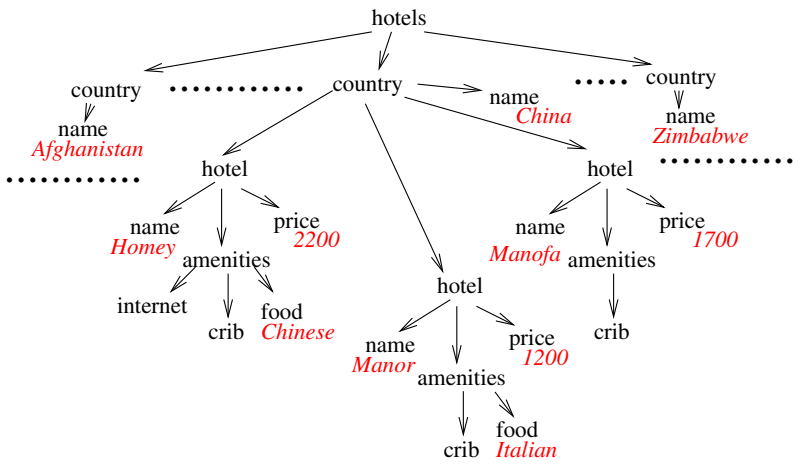
- The variables are divided into two sets: V_{req} containing *required* variables, that must have an image, and V_{opt} containing *optional* variables that may have no image.
- The edges are divided into two sets: E_{req} containing *required* edges that must be satisfied, and E_{opt} containing *optional edges* that may be unsatisfied.
- *Value orderings* are provided to state preferences over the textual content of the image of a variable. A *value ordering* for a variable v has the form $c_1 > \cdots > c_{k-1} > c_k$ where $c_i = v\,\theta_i s_i$, for all $i \leq k-1$ and $c_k = \mathbf{true}$.

In Section V we will consider extensions to our language of preferences, e.g., to allow preferences over labels of variables.

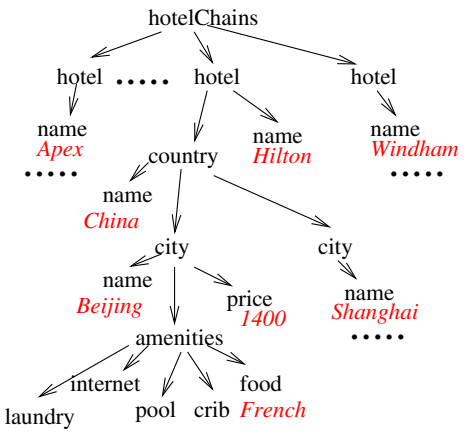
Definition 2.3 (Preference Query): A *preference query* P is a tuple $(V_{\mathsf{req}}, V_{\mathsf{opt}}, E_{\mathsf{req}}, E_{\mathsf{opt}}, C, O)$ where

- $(V_{\mathsf{req}} \cup V_{\mathsf{opt}}, E_{\mathsf{req}} \cup E_{\mathsf{opt}}, C)$ is an exact query;
- O contains a value ordering for each variable $v \in V_{\mathsf{req}} \cup V_{\mathsf{opt}}$.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.



Database D_1, grouped by country



Database D_2, grouped by hotel chain

Fig. 1. Databases D_1 and D_2, containing hotel information.

Throughout this paper, we will assume that each variable ordering is short, i.e., contains few constraints. We discuss the case in which $k-1$ may be very large in Section V.

Example 2.4: Sam's query Q_1 from Example 2.1 returned an empty answer over both D_1 and D_2, causing Sam to realize that his "ideal" hotel may not exist in China, and he must rethink his hotel requirements and desires.

Sam's new preference query P_1 appears in Figure 3. We use ? to indicate optional variables (i.e., variables from $V_{\mathsf{opt}}(P_1)$) and optional edges (i.e., edges from $E_{\mathsf{opt}}(P_1)$). The remainder of the variables and edges are from $V_{\mathsf{req}}(P_1)$ and $E_{\mathsf{req}}(P_1)$. Value orderings appear in blue italics, and are prepended with $?_i$ to indicate their relative ordering. We omit the constraint **true***, which appears in every value ordering, for brevity.*

In P_1 there are optional edges in both directions between hotel and country. This is used to capture both possible groupings. The Internet and star variables are now optional—because Sam is willing to stay at a hotel without Internet, and Sam does not mind if star information is not available. For hotels with star values, Sam prefers a 5-star hotel over a 4-star hotel, which, in turn, is preferred over a 3-star hotel. Sam needs food (and hence, food is in $V_{\mathsf{req}}(P_1)$), but has downgraded his desire for Chinese cuisine from a requirement to a preference. Finally, Sam requires a price below 3000. He would prefer the price to be below 1000 too, and if not, prefers a price below 2000. □

We will use $V_{\mathsf{opt}}(P)$, $V_{\mathsf{req}}(P)$, $E_{\mathsf{opt}}(P)$, $E_{\mathsf{req}}(P)$, $C(P)$ and $O(P)$ in the obvious way, to denote the appropriate part of P. In addition, we use $V(P)$ to denote $V_{\mathsf{opt}}(P) \cup V_{\mathsf{req}}(P)$ and $E(P)$ to denote $E_{\mathsf{opt}}(P) \cup E_{\mathsf{req}}(P)$.

A preference query P actually defines a set of exact queries, called *instantiations*, which may or may not contain optional variables and edges, and may contain additional constraints from O.

Definition 2.5 (Candidate Structure, Instantiation): A *candidate structure* for a preference query Q is a triple $X = (V, E, C)$ such that

- $V \subseteq V(P)$ and $V_{\mathsf{req}}(P) \subseteq V$, i.e., V is a subset of variables from P that contains all required variables;
- $E \subset E(P)$ and $E_{\mathsf{req}}(P) \cap (V \times V) \subseteq E$, i.e., E is a subset of the edges from P that contains all required edges (u, v) for which $u, v \in V$;
- C consists of
  1) all constraints from $C(P)$ that are defined over variables in V;
  2) A single constraint c_i from each value ordering in O of a variable in V.

An *instantiation* of P is a candidate structure of P that is also an exact query (i.e., is connected and rooted).

We use $\mathsf{Cand}(P)$ and $\mathsf{Inst}(P)$ to denote the set of all candidate structures and instantiations, respectively, of P.

Example 2.6: Three candidate structures for P_1 appear in Figure 4. The candidate structure X_1 is not an instantiation of P_1 since it is not connected. On the other hand, X_1 and X_2 are both candidates structure and instantiations of P_1. Observe that query Q_1 from Figure 2 is also an instantiation of P. □

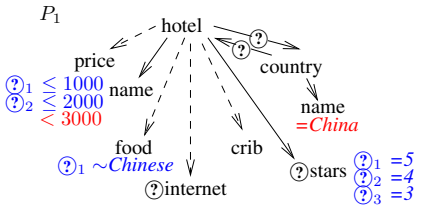


Fig. 3. Preference Query P_1

Now, given a database D and a preference query P, the

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.

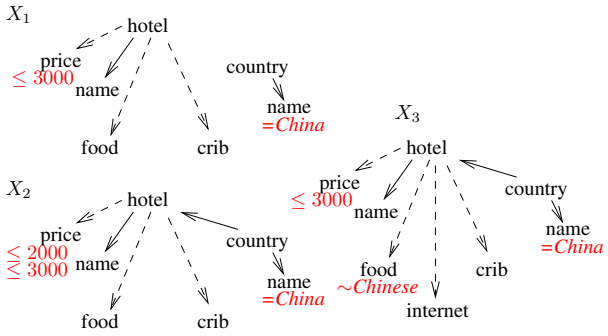


Fig. 4. Candidate Structures X_1, X_2 and X_3

result of applying P to D is the set $\mathsf{Ans}(P, D)$ defined as

$$\mathsf{Ans}(P, D) \ ::= \ \cup_{Q \in \mathsf{Inst}(P)} \mathsf{Ans}(Q, D)\,.$$

Note that the exact queries in $\mathsf{Inst}(P)$ may differ greatly one from another, in their sets of variables and edges. Some exact queries may contain very few of the optional variables and edges, and thus, may define answers that satisfy few of the conditions implied by P, even if these can be satisfied by the database. In the next section, we present the notion of "best" or *skyline* answers for P over D.

D. Skyline Answers

Let P be a preference query. We define a partial order $\succ$ over $\mathsf{Cand}(P)$. Since $\mathsf{Inst}(P) \subseteq \mathsf{Cand}(P)$, this also defines a partial order over $\mathsf{Inst}(P)$.

Definition 2.7 (Dominance): Let $X_1 = (V_1, E_1, C_1)$ and $X_2 = (V_2, E_2, C_2)$ be candidate structures of P. We say that X_2 *dominates* X_1, denoted $X_2 \succeq X_1$ if

- $V_2 \supseteq V_1$,
- $E_2 \supseteq E_1$, and
- for each variable $v \in Q_1$, if C_1 contains the constraint c_i from the value ordering of v, then C_2 contains a constraint c_j from the value ordering of v for which $j \leq i$;

We say that X_2 *strictly dominates* X_1, denoted $X_2 \succ X_1$ if $X_2 \succeq X_1$ and $X_2 \neq X_1$.

Intuitively, X_2 dominates since it contains all variables and edges of X_1, and contains preferable constraints over variables. The relationship $\succ$ is only a partial ordering, and many queries in $\mathsf{Inst}(P)$ are incomparable.

Example 2.8: Recall candidate structures Q_1, X_1, X_2 and X_3. It is not difficult to see that $X_2 \succ X_1$, since X_2 has all edges, nodes and constraints appearing in X_1. Similarly, observe that $X_3 \succ X_1$ and $Q_1 \succ X_1$. However, neither $Q_1 \succ X_2$ nor $X_2 \succ Q_1$ holds, since each has an edge not appearing in the other (between hotel and country). Likewise, there is no dominance between X_3 and Q_1 (because of the same structural reasons) nor between X_2 and X_3 (since X_2 is preferable on price and X_3 is preferable on food and Internet). □

We now use *skyline semantics* to define the "best" instantiations of P, as well as the "best" answers for P. Given a preference query P, a database D, and an exact query $Q \in \mathsf{Inst}(P)$, we say that Q is a *skyline instantiation* of P with respect to D, if

- $\mathsf{Ans}(Q, D) \neq \emptyset$ and
- there does not exist an exact query $Q' \in \mathsf{Inst}(P)$ such that $Q' \succ Q$ and $\mathsf{Ans}(Q', D) \neq \emptyset$.

The set of skyline instantiations for P with respect to D is denoted $\mathsf{SkyInst}(P, D)$.

The set of *skyline answers* for P over D is defined as

$$\mathsf{SkyAns}(P, D) \ ::= \ \cup_{Q \in \mathsf{SkyInst}(P,D)} \mathsf{Ans}(Q, D)$$

Example 2.9: Observe that $\mathsf{SkyInst}(P_1, D_1) = \{X_2, X_3\}$. Each of the queries X_2 and X_3 has a single answer over D_1. □

Our goal in this paper is to solve the following two problems.

Problem 1 (Skyline Instantiations): Given a database D and a preference P the *skyline instantiations* problem is to return $\mathsf{SkyInst}(P, D)$.

Problem 2 (Skyline Answers): Given a database D and a preference P the *skyline answers* problem is to return $\mathsf{SkyAns}(P, D)$.

The skyline answers problem is of interest in order to derive the "best" answers for the user's preference query. The skyline instantiations problem is of interest in order to perform automatic query correction, i.e., to automatically return to the user the set of "best" satisfiable exact queries for P. The user can choose among these to derive exactly the data of interest, in the spirit of [12].

III. Finding Skyline Instantiations and Results

We review related work on computing skyline answers over relational databases, and then present several variants of a solution for the skyline results and instantiations problems.

A. Related Work

Computation of skyline answers has been previously studied in the context of relational databases. In this context, one is given a set of tuples T, over some set of attributes $\mathcal{A}$. For a subset of attributes $\mathcal{B} \subseteq \mathcal{A}$, called *dimensions*, the user has predefined preferences. A preference for a dimension $B \in \mathcal{B}$ is given as a complete ordering $\geq_B$ over the domain $\mathsf{Dom}(B)$ of B. A tuple $t \in T$ strictly dominates $t' \in T$, if for all $B \in \mathcal{B}$,

$$t[B] \geq_B t'[B]$$

and there exists a $B_* \in \mathcal{B}$ such that

$$t[B_*] >_B t'[B_*]\,.$$

The problem of interest is then to find the set $\mathsf{SkyTuples}(T)$, containing all tuples $t \in T$ such that no tuple $t' \in T$ strictly dominates t.

Example 3.1: Consider the set of tuples T, appearing in Table I. Suppose that the user would prefer to have an Internet connection, prefers a larger number of stars, and a lower price. Thus, attributes Internet Connection, Stars and Price are

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.

	Hotel Name	Internet Connection	Stars	Price
t_1 :	Home Sweet	N	3	1050
t_2 :	Fancy Inn	Y	5	2200
t_3 :	Olden Hotel	N	3	1100

TABLE I
HOTEL INFORMATION

the dimensions of interest. Tuple t_3 is strictly dominated by t_1, since it has the same Internet Connection and Star values, but a worse value for Price. Therefore, t_3 is not in $\mathsf{SkyTuples}(T)$. *On the other hand, both t_1 and t_2 are in* $\mathsf{SkyTuples}(T)$, *as neither dominates the other (e.g., t_1 is preferable with respect to Price, but t_2 is preferable with respect to Stars).* □

Efficient computation of the skyline over relational data has been studied extensively, e.g., [1]–[6]. The algorithms developed differ in their assumptions about the number of dimensions, about the dimension cardinality (i.e., the number of values in the dimension domains) and about the data storage (e.g., presence or absence of index structures). However, these previous studies had a common assumption: *the tuples, from which the skyline must be chosen, are materialized.*

For relational data, the preferences are defined on a subset of the attributes. Thus, it is likely that there will be relatively few attributes of interest, each of which has high cardinality. Indeed, previous algorithms, other than [4], made exactly this assumption.

In [4], it was noted that many natural attributes have small domains, e.g., Stars in Example 3.1. Even attributes with large domains will often be mapped to a small number of ranges [4]. This is useful to avoid ruling out tuples that are simply slightly worse than a skyline tuple. For example, suppose that Price values in Example 3.1 were mapped in the following fashion: prices up until 1000 were mapped to "low", prices from 1000 to 2000 were mapped to "moderate," and prices above 2000 were mapped to "high". Then, tuple t_3 would remain in the skyline, which is natural since indeed it is not significantly worse than t_1.

The *lattice algorithm* was presented in [4] for preference queries with low dimension cardinality.[1] To find $\mathsf{SkyTuples}(T)$, the lattice algorithm creates a lattice $\mathcal{L}$ over all possible combinations of values in the domain of the dimensions $\mathcal{B}$. To be precise, suppose that $\mathcal{B} = \{B_1, \ldots, B_k\}$ and $\mathsf{Dom}(B_i)$ is the domain of B_i, for all i. Then, $\mathcal{L}$ is a lattice of k-tuples $\vec{b} \in \mathsf{Dom}(B_1) \times \cdots \times \mathsf{Dom}(B_k)$. There is a directed edge from $\vec{b}$ to $\vec{b}'$ in $\mathcal{L}$ if $\vec{b}$ strictly dominates $\vec{b}'$, and there is no $\vec{b}''$ such that $\vec{b}$ strictly dominates $\vec{b}''$ and $\vec{b}''$ strictly dominates $\vec{b}'$. Note that low cardinality implies that $\mathcal{L}$ is not prohibitably large.

[1]Actually, a single dimension with high cardinality was also allowed. This extension can also be carried over to the algorithms discussed in this paper, and is discussed in Section V.

The lattice algorithm starts by marking each node in $\mathcal{L}$ as **absent**. Then, the algorithm iterates over T. For each $t \in T$, it locates the node $\vec{b} \in \mathcal{L}$ such that t and $\vec{b}$ agree on all dimensions $\mathcal{B}$. The node $\vec{b}$ is marked as **present** and the tuple t is associated with $\vec{b}$. Finally, the algorithm traverses $\mathcal{L}$ in a top-down fashion. Each time that a node $\vec{b}$ marked **present** is found, the following two actions are taken: First, the tuples associated with $\vec{b}$ are printed. Second, all descendents of $\vec{b}$ are *pruned* from $\mathcal{L}$. It is easy to see that at the end of this process, exactly $\mathsf{SkyTuples}(T)$ will be printed.

In this paper, we develop an algorithm in the spirit of the lattice algorithm to solve the skyline instantiation and results problems for preference queries over XML. Such an algorithm is particularly appropriate in our setting, since most preferences have naturally low cardinality, e.g., optional variables and edges imply preferences of cardinality two (where satisfaction of the variable/edge is preferable to non-satisfaction). Our algorithm will be more intricate due to the fact that the common assumption discussed above (i.e., that the results to be chosen are already materialized) no longer holds in our problems.

B. The Candidate Lattice

Given a preference query P, we define the *candidate lattice* $\mathcal{L}(P)$. The lattice $\mathcal{L}(P)$ has a node X for each candidate structure (see Def. 2.5) of P. There is an edge from a X_1 to a node X_2 if $X_1 \succ X_2$ and there is no X_3 such that $X_1 \succ X_3 \succ X_2$.

Some of the nodes in $\mathcal{L}(P)$ are instantiations of P (since they are exact queries), while some are only candidate structures but not instantiations. Given a lattice node X, we will use $\mathsf{isInst}(X)$ to denote the boolean value indicating whether X is an instantiation.

Example 3.2: Figure 5 contains a preference query P_2, along with its candidate lattice $\mathcal{L}(P_2)$. Observe that nodes X'_1, X'_3, X'_4 and X'_7 are instantiations of P_2. The remaining nodes are only candidate structures for P_2, since they are disconnected.

Over D_1, the skyline instantiations for P_2 are $\mathsf{SkyAns}(P_2, D_1) = \{X'_3, X'_4\}$. □

We would like to use the candidate lattice $\mathcal{L}(P)$ in order to compute $\mathsf{SkyInst}(P, D)$ and $\mathsf{SkyAns}(P, D)$. Since the lattice edges follow the dominance relationship, we can find the set $\mathsf{SkyInst}(P, D)$ (and then afterwards $\mathsf{SkyAns}(P, D)$) by identifying the set of nodes $X \in \mathcal{L}(P)$ such that all the following conditions hold:

1) $\mathsf{isInst}(X) = \mathbf{true}$;
2) $\mathsf{Ans}(X, D) \neq \emptyset$;
3) there does not exist a node X' satisfying Requirements 1 and 2 such that there is a directed path from X' to X in $\mathcal{L}(P)$.

Requirement 1 ensures that all nodes X found are in fact instantiations. Requirement 2 ensures that X is an instantiation with a non-empty query result. Requirement 3 ensure that there is no instantiation that dominates X.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.

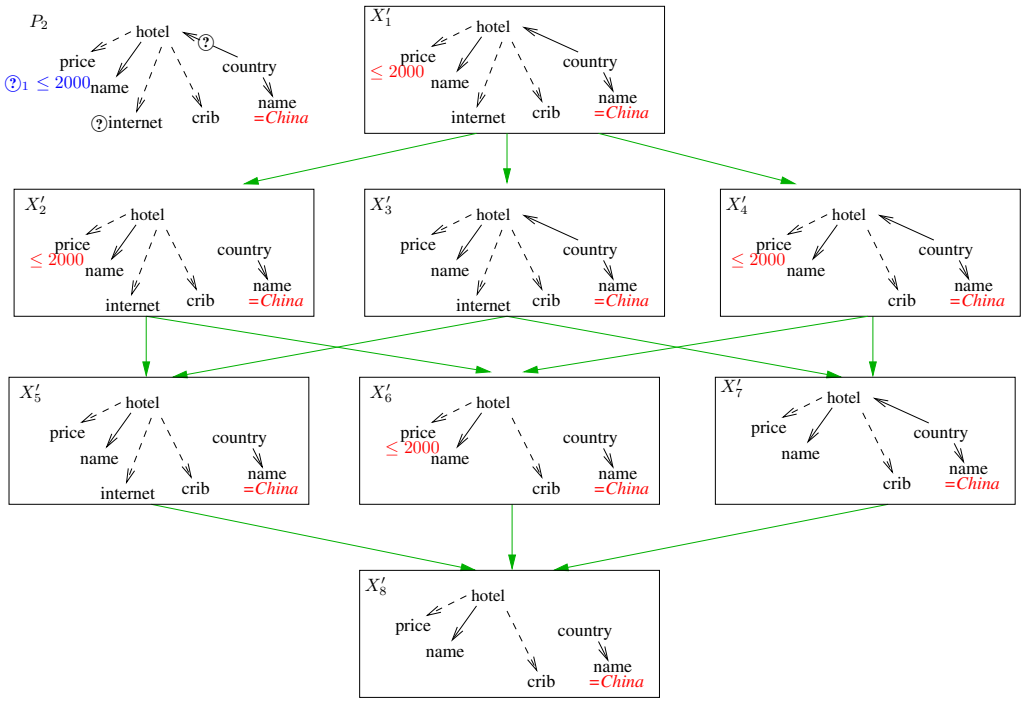


Fig. 5. Candidate Lattice for Preference Query P_2

Since the problems of finding $\mathsf{SkyInst}(P, D)$ and $\mathsf{SkyAns}(P, D)$ are very similar, we will focus on the latter, in the remainder of this paper. Experimental results for the problem of finding $\mathsf{SkyInst}(P, D)$ using (obvious adaptations of) the techniques for $\mathsf{SkyAns}(P, D)$ were similar to those for the problem of finding $\mathsf{SkyAns}(P, D)$, and are omitted due to lack of space.

We present three solutions for finding $\mathsf{SkyAns}(P, D)$. The first solution focuses on minimizing the number of queries that have to be evaluated. The second solution focuses on minimizing the number of intermediate results that are generated. The third solution is a hybrid of the first two solutions and strategically switches from the second to the first.

C. MinQuerySky: *Minimizing Number of Queries Evaluated*

Our first solution to the skyline answers problem is in the spirit of the lattice algorithm of [4]. The basic algorithm can be described as follows. We start by marking each node in $\mathcal{L}(P)$ as **absent**. We then, create all results in $\mathsf{Ans}(P, D)$. Next, we iterate over each result γ and locate all instantiations $X \in \mathcal{L}(P)$ such that $\gamma \in \mathsf{Ans}(X, D)$. Each of these instantiations is marked as **present** in $\mathcal{L}(P)$ and γ is associated with X. Finally, we traverse $\mathcal{L}(P)$ in a top-down fashion. Each time that a node X marked **present** is found, the following two actions are taken: First, the answers associated with X are printed. Second, all descendents of X are *pruned* from $\mathcal{L}(P)$. At the end of the process, we will have printed $\mathsf{SkyAns}(P, D)$.

Note that a single answer γ may be associated with several instantiations. For example, if $\gamma \in \mathsf{Ans}(Q_1, D)$ and Q_2 is an instantiation with the same variables and constraints, but less edges, then $\gamma \in \mathsf{Ans}(Q_2, D)$. Similarly, even if Q_2 has different constraints it may be possible for γ to be in its result. Storing each answer with each satisfying instantiation is quite wasteful in memory. It is sufficient to associate γ with the satisfying instantiations that are "highest" in the lattice. In other words, we associate γ with Q_1 if $\gamma \in \mathsf{Ans}(Q_1, D)$ and there is no Q_2 such that (1) $\gamma \in \mathsf{Ans}(Q_2, D)$ and (2) there is a directed path from Q_2 to Q_1.

We now consider the problem of computing $\mathsf{Ans}(P, D)$. Recall that

$$\mathsf{Ans}(P, D) = \cup_{Q \in \mathsf{Inst}(P)} \mathsf{Ans}(Q, D) .$$

Evaluating all queries in $\mathsf{Inst}(P)$ is very expensive. Instead, we would like to evaluate as few queries as possible, while still computing all of $\mathsf{Ans}(P, D)$.

To demonstrate how we can compute $\mathsf{Ans}(P, D)$, without evaluating all queries in $\mathsf{Inst}(P)$, we start by considering a special case.

Example 3.3: Suppose that $V_{\mathsf{opt}}(P) = \emptyset$, i.e., all variables in P are required. Let X be the candidate structure for P defined as $(V_{\mathsf{req}}(P), E_{\mathsf{req}}(P), C(P))$. In other words, X contains only the required variables and edges from P, as well as the constraints $C(P)$ (with the constraint **true** *chosen from each value ordering). Note that X is the minimal (lowest) node in the lattice $\mathcal{L}(P)$. Suppose that X is an instantiation, i.e., is an exact query. Then, it is not difficult to see that*

$$\mathsf{Ans}(X, D) = \mathsf{Ans}(P, D) .$$

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.

Thus the entire set $\mathsf{Ans}(P, D)$ *can be computed using a single query.* □

In general, more than one instantiation will be needed to generate all of $\mathsf{Ans}(P, D)$. Multiple queries are needed if at least one of the two assumptions made above do not hold: $V_{\mathsf{opt}}(P) \neq \emptyset$ or the candidate structure with only required edges is not an instantiation. We now precisely define a set of instantiations which yields all of $\mathsf{Ans}(P, D)$.

Let V_o be a subset of $V_{\mathsf{opt}}(P)$, and let $V_{o,\mathsf{req}}$ be the set of variables $V_o \cup V_{\mathsf{req}}(P)$. We use X_{V_o} to denote the candidate structure for P defined as $(V_{o,\mathsf{req}}, E_{\mathsf{req}}(P), C_{V_{o,\mathsf{req}}}(P))$, where $C_{V_{o,\mathsf{req}}}(P)$ is the restriction of C that contains constraints only for variables in $V_{o,\mathsf{req}}$. (Note that as before, this candidate structure actually chooses the constraint $\mathbf{true}$ from each value ordering.) Let $\mathcal{X}_{V_o}$ be all candidate structures X' such that $X' \succeq X_{V_o}$ and $V(X') = V(X_{V_o})$. Note that X_{V_o} is in the set $\mathcal{X}_{V_o}$ by definition. Finally, let $\mathcal{Q}_{V_o}$ be the subset of $\mathcal{X}_{V_o}$ that contains all $X \in \mathcal{X}_{V_o}$ such that

- $\mathsf{isInst}(X) = \mathbf{true}$, i.e., X is an instantiation of P;
- there is no $X' \in \mathcal{X}_{V_o}$ that is an instantiation of P, and $X \succ X'$.

Intuitively, $\mathcal{Q}_{V_o}$ contains all "lowest" instantiations of P with the variables $V_{o,\mathsf{req}}$. Note that $\mathcal{Q}_{V_o}$ may be empty, e.g., if the subgraph of P containing variables $V_{o,\mathsf{req}}$ is disconnected. Note also that every query in $\mathcal{Q}_{V_o}$ will have the same set of constraints, namely $C_{V_{o,\mathsf{req}}}(P)$.

We can prove the following result.

Lemma 3.4: Let P be a query. Then,

$$\mathsf{Ans}(P, D) = \bigcup_{\substack{V_o \subseteq V_{\mathsf{opt}}(P) \\ Q \in \mathcal{Q}_{V_o}}} \mathsf{Ans}(Q, D)\,.$$

Note that for the special case of Example 3.3, the union on the right side will contain a single query.

Example 3.5: Preference query P_2 has a single optional variable, namely, Internet. For the set $V_o = \emptyset$, the candidate structure X_{V_o} is X'_8 (Figure 5), $\mathcal{X}_{V_o} = \{X'_4, X'_6, X'_7, X'_8\}$ and $\mathcal{Q}_{V_o} = \{X'_7\}$, since X'_8, X'_6 are not instantiations and $X'_4 \succ X'_7$ (and hence, is not included). Similarly, for $V_o = \{\text{internet}\}$, $X_{V_o} = X'_5$, $\mathcal{X}_{V_o} = \{X'_1, X'_2, X'_3, X'_5\}$ and $\mathcal{Q}_{V_o} = \{X'_3\}$. Therefore, in order to compute $\mathsf{Ans}(P_2, D)$ over any database D, it is sufficient to compute the union of $\mathsf{Ans}(X'_7, D)$ and $\mathsf{Ans}(X'_3, D)$. □

To summarize, the strategy presented in this section to create $\mathsf{SkyAns}(P, D)$ is: First, generate $\mathsf{Ans}(P, D)$ using Lemma 3.4. Then, iterate over the results to assign them to lattice nodes. Finally, iterate over $\mathcal{L}(P)$ (top-down) to find all skyline results. We call this algorithm MINQUERYSKY, since the procedure focuses on minimizing the number of instances that must be evaluated.

D. MINRESSKY: Minimizing Number of Results Generated

Often, the algorithm MINQUERYSKY will evaluate few queries. Indeed, we have seen that it is possible that only a single query will be evaluated. However, MINQUERYSKY produces many answers to P that are not part of $\mathsf{SkyAns}(P, D)$. This is because MINQUERYSKY computes all of $\mathsf{Ans}(P, D)$ which may be *much* larger than $\mathsf{SkyAns}(P, D)$. In fact, $|\mathsf{Ans}(P, D)| \gg |\mathsf{SkyAns}(P, D)|$ should be the common case, since otherwise, the user could look at all of $\mathsf{Ans}(P, D)$ himself, in order to identify answers of interest.

The algorithm MINRESSKY only generates $\mathsf{SkyAns}(P, D)$, and no additional results. For this purpose, we create the lattice $\mathcal{L}(P)$. We traverse $\mathcal{L}(P)$ in a top-down fashion. Each time we reach a node X, we do the following. First, we check if $\mathsf{isInst}(X)$. If not, we can proceed immediately to the next node in our lattice traversal. If $\mathsf{isInst}(X) = \mathbf{true}$, then we evaluate $\mathsf{Ans}(Q, D)$. If this set is empty, once again, we proceed to the next node in the lattice traversal. If $\mathsf{Ans}(Q, D) \neq \emptyset$, then we print $\mathsf{Ans}(Q, D)$ and *prune* all descendents of X from $\mathcal{L}(P)$. At the end of the process, we will have printed $\mathsf{SkyAns}(P, D)$.

Example 3.6: When running MINRESSKY *on P_2 and D_1, we will start by evaluating X'_1. This query is unsatisfiable. Evaluation continues with X'_2, which is not an instantiation (and hence, is not evaluated). Next, X'_3 is evaluated, and yields a result. This result is printed and nodes X'_5, X'_7 and X'_8 are pruned, since they are descendents of X'_3. Finally, X'_4 is evaluated, and its result is printed, and X'_6 is pruned. The algorithm terminates, as all nodes in the lattice that have not been evaluated were pruned.* □

It is important to note that every query evaluated that is not part of $\mathsf{SkyInst}(P, D)$ is unsatisfiable over D. Hence, the only answers generated throughout the algorithm are exactly $\mathsf{SkyAns}(P, D)$. A distinct disadvantage of MINRESSKY is that we may end up evaluating many queries before finding the skyline instantiations. In fact, we may evaluate many queries which can not possibly yield any answers. A very simple example are cyclic queries, which are always empty. If we can determine whether a query is potentially satisfiable before evaluating over D, we can speed up MINRESSKY significantly. Determining satisfiability of queries is the topic of Section IV.

The algorithm MINRESSKY uses significantly less memory than MINQUERYSKY. In MINQUERYSKY, all of $\mathsf{Ans}(P, D)$ must be generated and stored. In MINRESSKY, no results need to be stored at all, since every result generated may be immediately printed to the output. Thus, the only memory requirements of MINRESSKY are for storage of the lattice $\mathcal{L}(P)$.

Normally, $\mathcal{L}(P)$ will fit conveniently into main memory and will not be prohibitably large. However, if P is very large, then $\mathcal{L}(P)$ may have an excessive size. Using MINRESSKY we can avoid the need to store $\mathcal{L}(P)$ if we generate the next nodes needed for traversal in $\mathcal{L}(P)$ on the fly, during the evaluation. Thus, using MINRESSKY even the entire $\mathcal{L}(P)$ does not need to be stored.

E. HYBRIDSKY: A Hybrid Approach

There is an inherent tradeoff between MINQUERYSKY and MINRESSKY. Usually, MINQUERYSKY will evaluate less queries than MINRESSKY. On the other hand, MINRESSKY

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.

will never create more results than MINQUERYSKY, and will usually create many less. A priori, it is not clear which algorithm will be faster for a particular database and query, as the relative speed depends heavily both on the number of results that MINQUERYSKY creates, as well as on the portion of the lattice that MINRESSKY must consider (i.e., the number and location of skyline instantiation nodes).

In order to get the best of both worlds, we propose the algorithm HYBRIDSKY. The algorithm HYBRIDSKY starts by *estimating* the amount of time $e(P, D)$ that MINQUERYSKY would run on the database D and preference query P. Then, HYBRIDSKY starts by running MINRESSKY. After each query evaluation during the pass over the lattice, HYBRIDSKY checks if the amount of time spent thus far is above a (predetermined) threshold percentage of the estimate $e(P, D)$. If this threshold has been surpassed, then HYBRIDSKY "switches mode" and runs the MINQUERYSKY algorithm until completion. To be more precise, let t be the amount of time spent in HYBRIDSKY at a given moment in its execution. Let $0 < p < 1$ be a predetermined threshold. If

$$t > p \cdot e(P, D) \tag{1}$$

then HYBRIDSKY switches to running MINQUERYSKY.

In Equation 1, the value p is predetermined, and the value t is measured during the runtime. It remains to show how the estimate $e(P, D)$ is derived. Before explaining this, we briefly discuss query evaluation in our system, since this must be understood to derive an estimate.

Each node in a database D is given an identifier. An identifier for a node n consists of a triple (s, f, l) where s is a number indicating the point in a DFS traversal over D which n is first reached, and f is a number indicating the point when the DFS traversal returns to n. Finally, l is the level of n. Such identifiers are well-studied, and they are useful since they allow the determination of whether a pair of nodes has an ancestor-descendent (or parent-child) relationship. For each label l appearing in D, we store the set $N_l(D)$ of all node identifiers for nodes labeled l in D. Query evaluation uses an array of size $|V(Q)|$, in which each cell contains database nodes that may match the query node corresponding to the cell. Evaluation proceeds using the standard bottom-up, top-down two pass algorithm (e.g., similar to that of Figure 8). During the bottom-up pass, we try to match each variable v with all nodes labeled $l(v)$. For each node $n \in N_{l(v)}(D)$, we check the matches already found for the children of v, to determine whether n can be used to satisfy the outgoing edges of v. In the top-down pass, we check for satisfiability of incoming edges for each v. Full details are omitted due to lack of space.

We now consider estimating the time of MINQUERYSKY. Our estimation $e(P, D)$ is based on the estimated actions taken by MINQUERYSKY:

- Query Evaluation: Let $\mathcal{Q}(P)$ be the set of queries needed to compute $\mathsf{Ans}(P, D)$ (Lemma 3.4). For each $Q \in \mathcal{Q}(P)$, the bottom-up pass of the query evaluation algorithm takes the following number of actions to find nodes matching to the variables (see the evaluation algorithm)

$$A(Q) = \sum_{v \in V(Q)} \left(|N_{l(v)}| \sum_{v' \in Children(v)} s_{v'} \times |N_{l(v')}| \right)$$

  where $s_{v'}$ is used to indicate the selectiveness of v' (i.e., due to its constraints or outgoing edges). The top down pass takes a similar number of actions.
- Placing Results on Lattice: Each result in $\mathsf{Ans}(Q, D)$ for $Q \in \mathcal{Q}(P)$ must be placed on the lattice. For each result, this takes time $Depth(\mathcal{L}(P))$, i.e., the depth of the candidate lattice (which is the sum of the number of optional variables, optional edges, and the sizes of the value orderings).
- Lattice Traversal: MINQUERYSKY traverses the lattice top-down to find the skyline instantiations. This takes time $|\mathcal{L}(P)|$.

In total, we estimate the time for MINQUERYSKY as

$$\frac{\sum_{Q \in \mathcal{Q}(P)} 2A(Q) + |\mathsf{Ans}(P, D)| \cdot Depth(\mathcal{L}(P)) + |\mathcal{L}(P)|}{pSpeed},$$

where $pSpeed$ is a constant used to translate the number of actions into time, by taking into consideration the processor speed. In our experimentation, $pSpeed = 10^{-4}$. Note that we still have to estimate s_v and $|\mathsf{Ans}(P, D)|$. Many previous works have studied estimating result sizes of XML queries, e.g., [13]–[16], and can be used to derive these values. Since estimating selectivity and result sizes was not the focus of this work, we used the simple value of 0.25 for s_v, and used the actual size of $|\mathsf{Ans}(P, D)|$. We expect that good estimations of s_v will improve the performance of HYBRIDSKY, and that using a (good) estimation of $|\mathsf{Ans}(P, D)|$ (instead of the actual value) will not adversely affect the performance of HYBRIDSKY.

IV. DETERMINING QUERY SATISFACTION

In the MINRESSKY and HYBRIDSKY algorithms, we may evaluate many instantiations before reaching the skyline instantiations. In MINRESSKY, all instantiations evaluated, other than the skyline instantiations, will derive empty results. This also holds in HYBRIDSKY, until the evaluation mode is switched to MINQUERYSKY.

Evaluating an instantiation against the database may be costly. Many evaluations may actually prove hopeless, since there may be some inherent problem with the query which implies that it will always return an empty answer, regardless of the database. For example, this is the case if the instantiation contains a cycle or contains a node with two incoming child edges, from variables with different labels. Since a database is a tree, the result of such queries must always be empty.

A query Q is *satisfiable* if there exists a database for which Q retrieves a non-empty result. We are often interested in databases with a restricted structure, e.g., as defined by a schema. Given a schema S, a query Q is satisfiable if there exists a database satisfying S for which Q returns a non-empty result. Satisfiability of exact queries (called tree patterns with

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.

joins) was studied in [17]. In the absence of a schema, a polynomial algorithm was presented to test for satisfiability. In the presence of an acyclic schema, the problem of determining satisfiability was shown to be NP-complete under combined complexity (when both the query and schema are part of the input).[2] Under query complexity (when the schema is assumed to be fixed size), satisfiability in the presence of an acyclic schema is solvable in polynomial time.

The algorithms for determining satisfiability, presented in [17], can be used to speed up the MINRESSKY and HYBRIDSKY algorithms, by first checking whether each query is satisfiable, before evaluating it over the database. In this section we present a new method of checking for satisfiability in the presence of a label-acyclic database *summary* that is polynomial under combined complexity. Thus, the result of this section is useful in the general context of query optimization (to check for satisfiability), beyond its usefulness for the skyline instantiation problem.

Intuitively, a *summary* is a graph that succinctly describes the database. We use the term summary, instead of schema, to emphasize the fact that a summary may be created from a database, instead of the standard assumption that a schema is first given, and a conforming database is created afterwards. In addition, our summaries may have multiple occurrences of the same label, to more exactly describe different contexts in which a label may be used.

Formally, a *summary* S is simply a directed, labeled graph. We use $N(S)$ to denote the nodes of S and $E(S)$ to denote the edges of S. We say that D *conforms* to S if there is a homomorphism φ from $N(D)$ to $N(S)$ such that

- $l(\varphi(n)) = l(n)$ and
- if $(n, m) \in E(D)$, then $(\varphi(n), \varphi(m)) \in E(S)$.

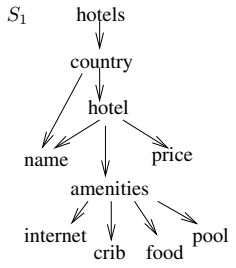


Fig. 6. Summary S_1 for database D_1

Let Q be an exact query and S be a summary. We say that Q is satisfiable with respect to S, denoted $\mathsf{Sat}(Q, S)$ if there exists a database D conforming to S such that $\mathsf{Ans}(Q, D) \neq \emptyset$. Given a database D, there are many summaries S such that D conforms to S. These summaries differ in their level of detail. For example, S_1 (Figure 6) is a summary for D_1.

Example 4.1: At one end of the spectrum, D is, itself, a summary of D. (This summary, of course, is not useful for speeding up satisfiability checks in our algorithms.) At the very other end of the spectrum, we define the summary S_{min}, containing a single node for each label l appearing in D. There is an edge (n, m) in S_{min} if there is an edge between nodes with labels $l(n), l(m)$ in D. The graph S_{min} is the smallest possible summary of D, and is often used as a simple abstraction of a schema, e.g. [18]. In the middle of the spectrum, one can use a graph index, e.g., Dataguide [19], 1-index [20], F&B-index [21]. □

[2]The query language considered in [17] is more expressive than our language, e.g., it can allow node inequalities.

Acyclic schemas were considered in [17]. In a similar vein we will consider only acyclic summaries. Moreover, as our summaries may contain multiple nodes with the same label (as opposed to the schemas of [17]), we will define a stronger notion of acyclicity.

Definition 4.2: We say that a summary S is *label acyclic* if there do not exist nodes $n, n', m, m' \in N(S)$, such that all the following conditions hold:

$$l(n) = l(n') \qquad (n, m) \in E^*(S)$$
$$l(m) = l(m') \qquad (m', n') \in E^*(S)$$

Intuitively, this means that there is a partial ordering $\prec_S$ of the labels appearing in S such that for any pair of nodes n, m, if n is an ancestor of m, then $l(n) \prec_S l(m)$. If S does not contain two nodes with the same label, then label-acyclicity coincides with the standard notion of acyclicity. Note that many databases have label-acyclic summaries, e.g., all databases conforming to a non-recursive DTD. The summary S_1, in particular, is label acyclic.

Remark 1: If the summary is not label-acyclic, then the methods for checking satisfiability, presented here, are not applicable. However, we can still use the satisfiability check from [17] for the schema-less case to speed up MINRESSKY.

Let Q be a query and S be a summary. We assume that Q contains only labels actually appearing in S. Otherwise, Q is trivially unsatisfiable. (This case is actually quite natural when the user is not sufficiently familiar with the database.) Now, in order to determine whether Q is satisfiable with respect to S, we will start by applying some rules to Q that will transform Q into a query Q', such that $\mathsf{Sat}(S, Q)$ if and only if $\mathsf{Sat}(S, Q')$. The rules will explicitly enforce constraints that are implicitly implied by the structure of Q and by S.

To perform this transformation, we repeatedly apply the following two rules, until no further applications are possible:

- *Rule 1:* If there are distinct variables $u, v \in Q$ such that (1) u and v have a common descendent, (2) $l(u) \neq l(v)$, and (3) there is no descendent edge from u to v or from v to u, then *add* $\mathsf{desc}(u, v)$ to Q if $u \prec_S v$, and $\mathsf{desc}(v, u)$ otherwise.
- *Rule 2:* If there are distinct variables $u, v \in Q$, that have a common descendent and $l(u) = l(v)$, then *merge* u with v. When merging u with v, we replace every edge involving v with an identical edge involving u, and replace every constraint involving v with an identical constraint involving u. Finally, we remove the variable v from Q completely.

For example, applying these rules to Q_2 from Figure 7 yields query Q_2'. Note that Q_2' has a new edge

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.

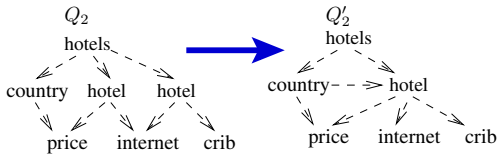


Fig. 7. Query Q_2, and result of rule applications Q'_2

```
FINDSATIMAGES(Q, S)
 1  A ← NEWMATRIX(|V(Q)|)
 2  for each u ∈ V(Q) in bottom-up order
 3       do A[u] = ∅
 4          for each n ∈ N(S) s.t. l(n) = l(u)
 5               do sat ← true
 6                  for each △(u, v) ∈ E(H)
 7                       do if ∄m ∈ A[v] s.t. S ⊨ △(n, m)
 8                            then sat ← false
 9                  if sat = true
10                    then A[u] ← A[u] ∪ {n}
11  for each u ∈ V(Q) in top-down order
12       do for each △(u, v) ∈ E(Q)
13               do for each m ∈ A[v]
14                       do if ∄n ∈ A[u] s.t. S ⊨ △(n, m)
15                            then A[v] ← A[v] − {m}
16  return A
```

Fig. 8. Procedure that returns a matrix associating each variable in Q with its satisfying images from S.

$\mathsf{desc}(\mathrm{country}, \mathrm{hotel})$ and that the two hotel variables were merged.

Proposition 4.3: Let Q be a query and S be a summary. Let Q' be the result of applying the two rules until no further applications are possible. Then, Q is satisfiable with respect to S if and only if Q' is satisfiable with respect to S.

Intuitively, Proposition 4.3 is correct because every database is a tree, and hence, if u and v have a common descendent, then in every answer to Q, the images of the nodes in u and v must lie on the same path (and their respective ordering is determined uniquely by S). If u and v have the same label, then these images must actually coincide, since S is label acyclic.

Now, assume that Q is the result of applying the two rules, until no further application is possible. We now consider three cases in which we can immediately determine that Q is unsatisfiable:

- If Q has a cycle, then Q is unsatisfiable (since all databases are trees).
- If Q contains variables u, v, w, such that w is a descendent of u and an ancestor of v, and moreover, Q contains an edge $\mathsf{child}(u, v)$, then Q is unsatisfiable. We determine that Q is unsatisfiable, since in all answers γ, we must have that $\gamma(u)$ is a parent of $\gamma(v)$, yet $\gamma(w)$ is a descendent of $\gamma(u)$ and an ancestor of $\gamma(v)$. This is clearly not possible.
- For each variable v, we consider the set of constraints involving v. If these constraints are not consistent (e.g., $\{v \leq 5, v \geq 7\}$), then Q is unsatisfiable. Since our constraints have a restricted form, consistency can easily be determined. We note that variable merges (in Rule 2) may cause constraints to be added to a variable, which can imply a contradiction at this point.

Finally, suppose that all three cases are not applicable. It can be shown that the query derived thus far is equivalent to a tree. Therefore, we can perform a final satisfiability check for Q uses a two-pass algorithm to create a matrix $\mathcal{A}$.[3] This matrix will contain, for each $v \in V(Q)$, the set of *satisfying images* from S. Formally, we say that a node $n \in N(S)$ is a satisfying image for v if there exists a database D conforming to S, a homomorphism φ from D to S, and an answer $\gamma \in \mathsf{Ans}(Q, D)$ such that $\varphi(\gamma(v)) = n$.

The procedure FINDSATIMAGES (Figure 8) returns a matrix that associates each $v \in V(Q)$ with its set of satisfying images from S. In this algorithm, we use the notation $\triangle(u, v)$ as an abstract notation for an edge (that does not explicitly state whether the edge is a child or descendent edge). We use $S \models \triangle(n, m)$ if $\triangle = \mathsf{child}$ and $(n, m) \in E(S)$, as well as if $\triangle = \mathsf{desc}$ and $(n, m) \in E^*(S)$.

Proposition 4.4: Let $\mathcal{A} = \text{FINDSATIMAGES}(Q, S)$. Then, for each $v \in V(Q)$, the cell $\mathcal{A}[v]$ contains exactly the set of satisfying images for v in S.

Now, it immediately follows that Q is satisfiable if and only if there is no cell in $\mathcal{A}$ that is empty. (Actually, it is sufficient to check whether the cell for the root of Q is empty.)

We conclude the following result.

Lemma 4.5: Let Q be a query and S be a label-acyclic summary. Then, it is possible to determine whether Q is satisfiable with respect to S in polynomial time under combined complexity (where Q and S are both part of the input).

V. EXTENSIONS

In this section, we briefly consider several extensions to our language for preference queries.

More Expressive Preference Queries: We can increase the expressiveness of our preference queries. For example, we can allow preferences over the labels of a variable, instead of each variable having a predefined label. Note that our satisfiability check will still be correct, since each query tested for satisfiability will have a single label. However, the set of queries that must be evaluated for MINQUERYSKY will change, since we must now consider nodes that have a disjunction of labels on their variables. This only requires a small adaptation to our query evaluation algorithm

Another change of interest is to allow variables to be unlabeled (i.e., wildcard variables). No significant changes are needed to MINQUERYSKY, MINRESSKY or HYBRIDSKY to deal with the change. However, our satisfiability check will

[3]This algorithm is correct only since Q is equivalent to a tree. If Q is simply acyclic, but not a tree, then a mapping from Q to S might not imply that there may be a mapping from Q to a database conforming to S. The problem stems from the fact that S is an acyclic graph, whereas all conforming databases must be trees. Further details are omitted due to lack of space.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.

no longer be applicable. Instead, we can use the satisfiability check from [17] for the schema-less case.

Finally, we can increase the language of our constraints to allow arbitrary conjunctions and disjunctions of constraints (both directly on nodes, and as part of value orderings). While our algorithms remain the same, our satisfiability check can no longer be applied. Indeed, such a change renders the satisfiability problem NP-complete.

Preferences Orderings with Large Cardinality: As mentioned in [4], a slight adaptation to the lattice algorithm allows the inclusion of one dimension with a large domain cardinality. Specifically, if B has a large domain cardinality, then before a result is placed at a lattice node, it is first compared with all results already appearing at the node. If the new result has a preferable value for B, then all previous results are removed from the node and the new result is added. Conversely, if the new result has a less preferable value for B, then it is simply discarded. Finally, if the new result has an equally preferable value for B as those at the node, then it is simply added to the node. Using a similar adaptation to our algorithms, we can also allow a single preference with a large domain cardinality, e.g., "Price" for our examples.

Finding "Next-Best" Results: Skyline semantics returns the "best" results for a given query. However, if there are not enough results of this type, the user may be interested in viewing the "next-best" results. Two make this precise, we define the notion of 2nd-level, and, more generally i-level, skyline instantiations and answers.

Let P be a preference query and D be a database. We define i-level skyline instantiations, denoted $\mathsf{SkyInst}_i(P, Q)$, as follows. First, $\mathsf{SkyInst}_1(P, D)$ is simply $\mathsf{SkyInst}(P, D)$. For $i > 1$, we define $\mathsf{SkyInst}_i(P, D)$ to be all queries Q in $\mathsf{Inst}(P)$ such that for all $Q' \succ Q \in \mathsf{Inst}(P)$, we have $Q' \in \mathsf{SkyInst}_j(P, D)$ for some $j < i$. In other words, instantiations in $\mathsf{SkyInst}_i(P, D)$ are only dominated by instantiations in lower (numerical) levels of the skyline. Similarly, i-level results, denoted $\mathsf{SkyAns}_i(P, D)$ are defined as

$$\mathsf{SkyAns}_i(P, D) \ ::= \ \cup_{Q \in \mathsf{SkyInst}_i(P,D)} \mathsf{Ans}(Q, D)\,.$$

A new problem of interest can be defined as follows.

Problem 3 (Skyline-Order Result Enumeration): Given a database D, a preference query P, and integer k, the skyline-order result enumeration problem is to return

$$\mathsf{SkyAns}_1(P, D), \cdots, \mathsf{SkyAns}_k(P, D)$$

in an order such that for all $i < k$ all answers in $\mathsf{SkyAns}_i(P, D)$ should be returned before any answers of $\mathsf{SkyAns}_{i+1}(P, D)$.

All three of our algorithms can be used to solve the skyline-order result enumeration problem, by a simple adaptation that continues the execution after the skyline results have been generated. Basically, all that is needed is to continue the lattice traversal until the kth-level results are found. The results will naturally be found in their level order, and hence, will be printed in the correct order.

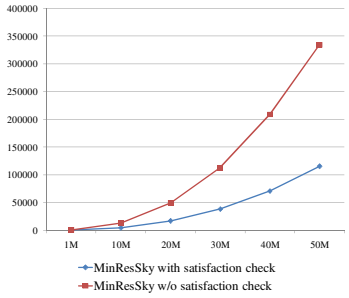


Fig. 9. Testing satisfiability check improvement.

VI. Experimentation

All three algorithms were implemented in Java on a personal computer running Windows XP. Data was created using XMark [22].[4] We discuss the results of our experimentation. All times are in milliseconds.

Satisfiability Check: Figure 9 compares the running time of MinResSky with and without the check for satisfiability against a summary. The experiment shows that satisfiability checks improve the runtime increasingly, as the size of the database increases. This is expected, since query evaluation is a function of the database size. Hence, avoiding query evaluation by satisfiability checks, becomes increasingly significant.

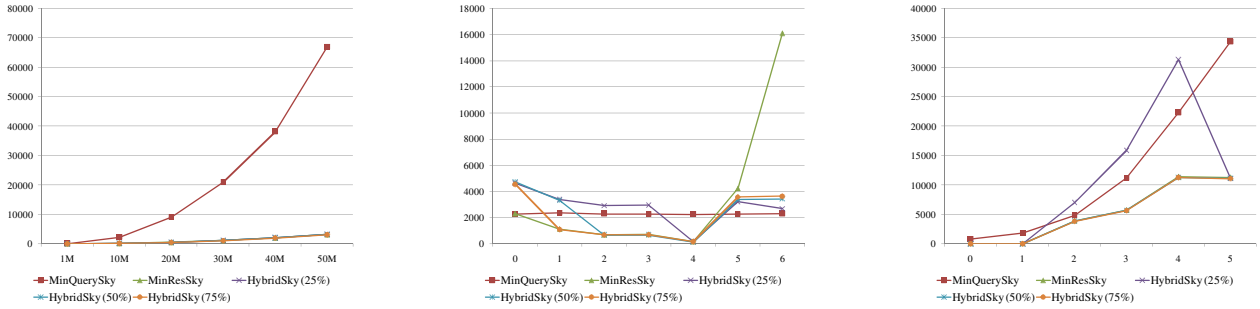
Increasing Database Size: Figure 10(a) compares the running times of MinQuerySky, MinResSky and three versions of HybridSky with varying thresholds p (0.25, 0.5, 0.75). The algorithm MinQuerySky degrades as the database size increase from 10 megabytes to 50 megabytes due to the increasing number of results that must be processed. All three hybrid versions coincided with MinResSky, since they never reached the threshold for switching modes to MinQuerySky.

Increasing Query Size: Since MinResSky is clearly superior to MinQuerySky on large databases, we focused on a moderately sized database of size 10 megabytes, to compare running time with increasing query size. Figure 10(b) shows the effect of an increasing number of optional edges. MinQuerySky is not adversely affected, since it evaluates the same set of queries to generate all results. On the other hand, MinResSky suffers from a decrease in performance, due to the need to explore a much larger lattice. The three versions of HybridSky perform similarly one to another, but vary on when they switch modes to MinQuerySky.

Figure 10(c) shows the effect of an increasing number of optional nodes. In this case, MinQuerySky degrades, due to the need to evaluate many additional queries, while the change in time for MinResSky is moderate. HybridSky with thresholds of 0.5 and 0.75 coincide with MinResSky, while HybridSky with threshold of 0.25 switches too early to MinQuerySky, thus, paying a running time penalty.

[4]XMark does not have a completely label-acyclic summary. In order to apply our satisfaction check improvements, our queries considered a label-acyclic portion of the documents.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.



(a) Increasing database (b) Increasing optional edges (c) Increasing optional nodes

Fig. 10. Experimental comparison of algorithms.

Neither MINQUERYSKY nor MINRESSKY is clearly superior one to another for moderate sized databases. However, HYBRIDSKY (with a threshold of 0.75) gives a "best of both worlds" performance.

VII. CONCLUSION

We presented a language for flexible querying of XML, that allows preferences over both the data values and over the structure. Skyline semantics are used to find optimal answers to a preference query. Three different algorithms were presented for query evaluation. Experimentation showed algorithm efficiency, particularly for HYBRIDSKY.

Future work includes incorporating estimations for constraint selectivity and result sizes directly into the system, as well as additional methods of checking for query satisfiability. It is also important to do user testing to experimentally compare our approach with previous score-based approaches for inexact querying of XML.

ACKNOWLEDGMENT

This work was partially supported by the ISF (Grant 1032/05).

REFERENCES

[1] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting: Theory and optimizations," in *Intelligent Information Systems*, 2005, pp. 595–604.
[2] P. Godfrey, R. Shipley, and J. Gryz, "Algorithms and analyses for maximal vector computation," *VLDB J.*, vol. 16, no. 1, pp. 5–28, 2007.
[3] K.-L. Tan, P.-K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *VLDB*, 2001, pp. 301–310.
[4] M. D. Morse, J. M. Patel, and H. V. Jagadish, "Efficient skyline computation over low-cardinality domains," in *VLDB*, 2007, pp. 267–278.
[5] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries," in *VLDB*, 2002, pp. 275–286.
[6] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.
[7] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit, "Flexpath: Flexible structure and full-text querying for xml," in *SIGMOD*, 2004, pp. 83–94.
[8] S. Amer-Yahia, S. Cho, and D. Srivastava, "Tree pattern relaxation," in *EDBT*, 2002, pp. 496–513.
[9] B. Kimelfeld and Y. Sagiv, "Combining incompleteness and ranking in tree queries," in *ICDT*, 2007, pp. 329–343.
[10] Y. Kanza and Y. Sagiv, "Flexible queries over semistructured data," in *PODS*, 2001.
[11] Y. Kanza, W. Nutt, and Y. Sagiv, "Querying incomplete information in semistructured data," *J. Comput. Syst. Sci.*, vol. 64, no. 3, pp. 655–693, 2002.
[12] T. Brodianskiy and S. Cohen, "Self-correcting queries for xml," in *CIKM*, 2007, pp. 11–20.
[13] Y. Wu, J. M. Patel, and H. V. Jagadish, "Using histograms to estimate answer sizes for xml queries," *Inf. Syst.*, vol. 28, no. 1-2, pp. 33–59, 2003.
[14] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang, "Statistical learning techniques for costing xml queries," in *VLDB*, 2005, pp. 289–300.
[15] C. Luo, Z. Jiang, W.-C. Hou, F. Yan, and C.-F. Wang, "Estimating xml structural join size quickly and economically," in *ICDE*, 2006.
[16] N. Zhang, M. T. Özsu, A. Aboulnaga, and I. F. Ilyas, "Xseed: Accurate and fast cardinality estimation for xpath queries," in *ICDE*, 2006.
[17] L. V. S. Lakshmanan, G. Ramesh, H. Wang, and Z. J. Zhao, "On testing satisfiability of tree pattern queries," in *VLDB*, 2004, pp. 120–131.
[18] S. Cohen, Y. Kanza, B. Kimelfeld, and Y. Sagiv, "Interconnection semantics for keyword search in xml," in *CIKM*, 2005, pp. 389–396.
[19] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," in *VLDB*, 1997, pp. 436–445.
[20] T. Milo and D. Suciu, "Index structures for path expressions," in *ICDT*, 1999, pp. 277–295.
[21] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, "Covering indexes for branching path queries," in *SIGMOD*, 2002, pp. 133–144.
[22] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "Xmark: A benchmark for xml data management," in *VLDB*, 2002, pp. 974–985.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on March 22,2010 at 08:05:55 EDT from IEEE Xplore. Restrictions apply.