Materialized View Selection in XML Databases

Nan Tang[†] Jeffrey Xu Yu[‡] Hao Tang[◇] M. Tamer Özsu[§] Peter Boncz[†]

[†] CWI, Amsterdam, The Netherlands email:{N.Tang,P.Boncz}@cwi.nl
[‡] The Chinese University of Hong Kong, Hong Kong email:yu@se.cuhk.edu.hk
[◇] Renmin University, Peking, China email:haotang@ruc.edu.cn
[§] University of Waterloo, Ontario, Canada email:tozsu@cs.uwaterloo.ca

Abstract. Materialized views, a RDBMS silver bullet, demonstrate its efficacy in many applications, especially as a data warehousing/decison support system tool. The pivot of playing materialized views efficiently is view selection. Though studied for over thirty years in RDBMS, the selection is hard to make in the context of XML databases, where both the semi-structured data and the expressiveness of XML query languages add challenges to the view selection problem. We start our discussion on producing minimal XML views (in terms of size) as candidates for a given workload (a query set). To facilitate intuitionistic view selection, we present a view graph (called VCUBE) to structurally maintain all generated views. By basing our selection on VCUBE for materialization, we propose two view selection strategies, targeting at space-optimized and space-time tradeoff, respectively. We built our implementation on top of Berkeley DB XML, demonstrating that significant performance improvement could be obtained using our proposed approaches.

1 Introduction

Materialized views, a RDBMS silver bullet, increase by orders of magnitude the speed of queries by allowing pre-computed summaries. In the context of XML databases, both the semi-structured data and the expressiveness of XML query languages complicate the selection of views. Answering XML queries using materialized views [4, 7, 15, 16, 22, 24] and XML view maintenance [20, 21] have been addressed recently, however, the problem of materialized view selection, which plays an important role in profiting from the above notable achievements of answering XML queries using views, is not well discussed. In this work, we describe a framework of selecting XML views to materialize. Broadly speaking, the problem of view selection in XML databases is the following: given XML databases $\mathcal{X}$, storage space B and a set of queries $\mathcal{Q}$, find a set of views $\mathcal{V}$ over $\mathcal{X}$ to materialize, whose combined size is at most B.

The problem of view selection is well studied in on-line analytical processing (OLAP) in data warehouses [5, 6, 9, 10, 13, 25], where multiple aggregated views are organized as a data cube [11, 18] for pre-computed summaries. This problem is generalized to network environments e.g. distributed databases [14] and P2P networks [8], as both computation cost and net communication could be saved.

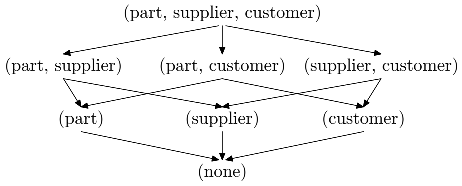


Fig. 1. Sample Data Cube

Q_1	//`conference[@booktitle = "SIGMOD" or "VLDB"]`
	`/author[@name = "Venky Harinarayan"`
	`or "Jeffrey D. Ullman"]`

Q_2	//`conference[@keyword = "view" or "data cube"]` `/author[@name = "Jeffrey D. Ullman"]`
Q_3	//`conference[[@keyword = "view" or "data cube"]` `and[@booktitle = "SIGMOD" or "VLDB"]]`

Table 1. Sample XML Queries

XML is a new standard for data exchange over the Internet, with standardized query languages (e.g. XPATH and XQUERY). There are similarities, and differences (in syntactic level) for the *view selection* problem between XML and relational databases: (*i*) there is inherent impedance mismatch between the relational (sets of tuples) and XML (ordered tree) data model; (*ii*) SQL is for relational data which is flat, regular, homogeneous and unordered. XPATH and XQUERY are for XML data which is nested, irregular, heterogeneous and ordered. Both items add to the complexity of the view selection problem.

We consider fundamental properties of view selection for optimizing a given query workload, with the objective in accordance with the formal perspective of view selection in RDBMS [6]. There are two conventional solutions in RDBMS: *common subexpressions exploitation* and *query aggregation.* The main problem of adopting common subexpressions is that the *deep copied* XML fragments lose the structural information required for performing the subsequent structural joins. While in RDBMS, the relational algebra is straightforwardly operated on intermediate tuples.

Next we discuss whether *query aggregation* can be applied. We review this technique in RDBMS first. Consider a TPC-D database with three tables: *part, supplier* and *customer.* The 8 possible groupings of tables are depicted in Figure 1. Computing each of the 8 groups of tables can be derived from the cells that are reachable to it, e.g., (*part*) can be derived from (*part, supplier*), (*part, customer*) and (*part, supplier, customer*). There are two difficulties of applying this technique in XML, considering the three XML queries in Table 1. (1) Enumerating all views that may contribute to answering a query is expensive. Assume that, given a query, we can find all views to answer it by relaxing queries [3]. For example, relax query axis e.g. //`conference/author` to //`conference`//`author` in Q_1 and Q_2; generalize label to wildcard $*$ e.g. //`conference` to //$*$ in Q_{1-3}, etc. The number of queries in this problem space is prohibitively large, which makes it infeasible. (2) Given candidate views, what relationships among them are to be maintained and in which way to organize them such that efficient algorithms might be devised, like data cube for RDBMS. Note that query/view answerability is closely related to query/view containment [16,17]. On the surface, it is natural to maintain the containment relationship between views. However, checking containment of XPATH queries is coNP-complete [17], with only a restricted syntax set $(/, //, *$ and branches $[\cdots])$. This indicates that it is impractical to find all containment relationships among candidate views.

Contributions & Roadmap. We start with an introduction of XML query and problem statement in Section 2. In Section 3, we prove the existence and uniqueness of a minimal view to answer two given queries, followed by an algorithm to generate such a view. We then describe, given multiple queries, how to generate a minimal view set while ensuring optimality. This solves the first difficulty. Note that computing the containment relationship between all candidate views ($\mathcal{V}$) requires $\mathsf{P}_2^{|\mathcal{V}|}$ (order 2 permutation of $|\mathcal{V}|$) comparisons. In Section 4, we propose to maintain the minimal view set using a directed acyclic graph, called VCUBE, to solve the second difficulty. VCUBE maintains the answerability relationship between views as edges, along with the process of generating views. This avoids computing the relationship between views pairwise, which is expensive. In Section 5, we describe two algorithms on top of VCUBE to select a set of views to materialize. Moreover, extensive experiments are conducted in Section 6 to show the efficiency of our proposed algorithms in boosting query performance. We conclude this paper in Section 7 with outlook on future works.

2 XML View Selection

2.1 Preliminaries

XPATH ***query.*** The set of XPATH queries studied in this paper is given in Table 2. It may involve the child-axis (/), descendant-axis (//), wildcards ($*$) and predicates. Predicates can be any of these: equalities with string, comparisons with numeric constants, or an arbitrary XPATH expression with optional operators "`and`" and "`or`". Sample XPATH queries may reference to Table 1.

Query containment. The *containment* of XPATH fragments involving /, //, $*$ and [] is studied in [17]. The extension of XPATH fragments containing additional operators "`and`" and "`or`" in predicates is studied in [4]. We borrow the definition of *containment* of queries from [17]. For an XML query P, $P(\mathcal{X})$ denotes the *boolean* result of P over database $\mathcal{X}$. We say $P(\mathcal{X})$ true if there exists at least one result; it is false otherwise. For two queries P and Q, P is *contained* in Q, denoted as $P \sqsubseteq Q$, iff $P(\mathcal{X})$ implies $Q(\mathcal{X})$, in every XML database $\mathcal{X}$. Containment is a partial order, i.e., $V \sqsubseteq P, P \sqsubseteq Q \Rightarrow V \sqsubseteq Q$. Equivalence is a two-way containment. Two queries P, Q are equivalent, denoted as $P \equiv Q$, iff $P \sqsubseteq Q$ and $Q \sqsubseteq P$.

```
      Path ::= Step+
      Step ::= Axis NameTest Predicates?
      Axis ::= "/" | "//"
  NameTest ::= elementName | " * "
Predicates ::= "[" Expr "]"
      Expr ::= Step | OrExpr
    OrExpr ::= AndExpr | OrExpr "or" AndExpr
   AndExpr ::= CompExpr | AndExpr "and" CompExpr | Step
  CompExpr ::= "@" attributeName Comp Value
      Comp ::= " = "|" > "|" < "|" >= "|" <= "|"!= "
     Value ::= Number | String
```

Table 2. Query Definition

Example. (1) $//a/b$ is contained in $//a//b$, since the //-axis is more general than /-axis; (2) $//a/b$ is contained in $//a/*$, since the wildcard $*$ matches any label; (3) $//a[@n < 10]$ is contained in $//a[@n < 100]$; (4) $//a[@s = \text{“}x\text{”}]$ is contained in $//a[@s = \text{“}x\text{”}\ or\ @s = \text{“}y\text{”}]$.

Answering queries using views. Existing works focus on rewriting a given query using materialized view with/without accessing the base data. We attempt to use the materialized view only to answer a query, without accessing the base data. $V \vDash Q$ denotes that a view V can be used to answer a query Q. $V \vDash \mathcal{Q}$ denotes that a view V can be used to answer each query in $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_n\}$.

2.2 Problem Statement

XML ***view selection.*** The problem of XML view selection is formally defined as follows: let $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_n\}$ (each Q_i is associated with a non-negative weight w_i) be a workload, $\mathcal{X}$ be an XML database, B be the available storage space, and COST() be a cost estimation function for query processing. The problem is to find a set of views $\mathcal{V}$ whose total size is at most B that minimizes:

$$\text{COST}(\mathcal{X}, \mathcal{V}, \mathcal{Q}) = \sum_{Q_i \in \mathcal{Q}} \text{COST}(\mathcal{X}, \mathcal{V}, Q_i) \times w_i \qquad (1)$$

here, $\text{COST}(\mathcal{X}, \mathcal{V}, Q_i)$ denotes the cost of evaluating query Q_i using some view in $\mathcal{V}$, which is materialized over the XML database $\mathcal{X}$. This object function is in accordance with the formal perspective of view selection in RDBMS [6].

The optimal solution. For a query Q_1, the *complete* set of candidates $\mathcal{C}_1$ is all views V that may answer Q_1, i.e., $\mathcal{C}_1 = \{V | V \vDash Q_1\}$. The complete set of candidates $\mathcal{C}$ for a workload $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_n\}$ is $\mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i$. The naïve way is to enumerate all candidate views $\mathcal{C}$. Then, by exhaustively searching $\mathcal{C}$ to identify the optimal solution (i.e. a view set $\mathcal{O}$) that minimizes Equation 1.

There is no existing work to enumerate all complete candidate views. Assume that for a query Q, we can define a set of rules, such as an edge relaxation (e.g., $/a$ to $/\!/a$), a subtree promotion (e.g., $/\!/a/b[Q_1]/\!/Q_2$ to $/\!/a[./\!/Q_2]/b[Q_1]$), a leaf node deletion (e.g., $/\!/a/b/c$ to $/\!/a/b$) [2], and label generation (e.g., $/\!/a$ to $/\!/*$), etc. With above rules, we may (possibly) enumerate the complete candidate views, while we still face the two difficulties addressed in Introduction. (1) The number of candidate views is prohibitively large. The basic relaxation rules [2] already generate an exponential amount of candidates. Taking label generation into account, the problem space is exponentially exploded. (2) It is infeasible to identify the relationships among all candidate views, which requires $\mathsf{P}_2^{|\mathcal{C}|}$ comparisons, and per containment check is NP-hard. We describe a new way to generate a small view set $\mathcal{V}$, which is a subset of the complete set (i.e., $\mathcal{V} \subseteq \mathcal{C}$) but ensures to be a superset of the optimal solution (i.e., $\mathcal{O} \subseteq \mathcal{V}$).

3 Candidate View Generation

We first discuss the problem of answering a query using materialized views only, i.e., without accessing the base data. We then describe how to generate a minimal set of views as candidates for materialization.

3.1 Query/View Answerability Criteria

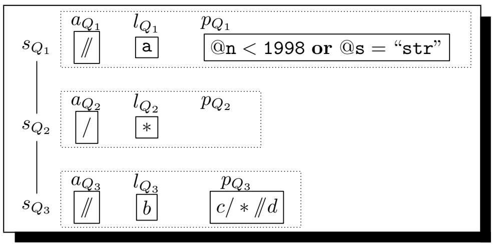


Fig. 2. Query Steps

Recall that in Table 2, each query Q is represented as a sequence of location steps $Q = s_{Q_1}s_{Q_2}\cdots s_{Q_n}$. Here, each step s_{Q_i} has the form $a_{Q_i}l_{Q_i}[p_{Q_i}]$, where $a_{Q_i} \in \{/, /\!/\}$ is an `Axis`, l_{Q_i} is an element name or wildcard $*$ for `NameTest`, and p_{Q_i} is a predicate that can be any XPATH fragment or empty. Figure 2 shows the three steps of query $Q : /\!/a[@n < 1998$ or $@s = \text{“}str\text{”}]/*/\!/b[c/*/\!/d]$. For materialization, the XML fragments that satisfy the query and rooted at the label of the last step (i.e. b) will be materialized as *deep copies* of XML fragments.

Criteria for answerability. Given a view V and a query Q, where $V : s_{V_1}s_{V_2}\cdots s_{V_m}$ and $Q : s_{Q_1}s_{Q_2}\cdots s_{Q_n}$, V answers Q, denoted as $V \vDash Q$, iff the following conditions are satisfied: (1) $m \leq n$; (2) $s_{Q_i} \equiv s_{V_i}$ (equivalent) for $1 \leq i \leq m-1$; and (3) $s_{Q_m}s_{Q_{m+1}}\cdots s_{Q_n} \sqsubseteq s_{V_m}$.

Item 1 states that the number of view steps must be no more than the number of query steps. Item 2 declares that the first $m-1$ query steps must be exactly the same as corresponding view steps. Therefore, we need not access the base data to refine the materialized view fragments. Two steps are equivalent, denoted as $s_{Q_i} \equiv s_{V_i}$, iff $a_{Q_i} = a_{V_i}$ (the same axis), $l_{Q_i} = l_{V_i}$ (the same label) and $p_{Q_i} \equiv p_{V_i}$ (equivalent predicates). Though being NP-hard for testing the equivalence/containment of two predicates in theory, the predicates are typically not complicated such that it could be handled in real applications. Item 3 claims that the XPATH fragment with the form $s_{Q_m}s_{Q_{m+1}}\cdots s_{Q_n}$ is contained in the XPATH fragment s_{V_m}, which guarantees that the query result can be extracted from the materialized view result. This criteria are similar to the ones used in [16].

3.2 Constructing A Minimal Query

Given two queries Q_1 and Q_2, we say Q is a minimal query that answers Q_1 and Q_2, iff $Q \vDash Q_1, Q \vDash Q_2$, and there does not exist another query Q' where $Q' \vDash Q_1, Q' \vDash Q_2$ and $Q' \vDash Q$. We have the following theorem.

Theorem 1. *Given two queries P, Q, the minimal query that answers both P and Q exists and is unique.*

Proof. There exists a query $/*$ (abbreviation of /child::$*$) that answers both P and Q. The query $/*$ actually materializes the very first *root* element of an XML document, which carries the entire information of an XML document (the virtual *document root* is exclusive).

Next we prove by construction that there exists a minimal query V, where $V \vDash P$, $V \vDash Q$. For any query V', if $V' \vDash P$ and $V' \vDash Q$, then $V' \vDash V$. We depict the view V and two queries P, Q in Figure 3.

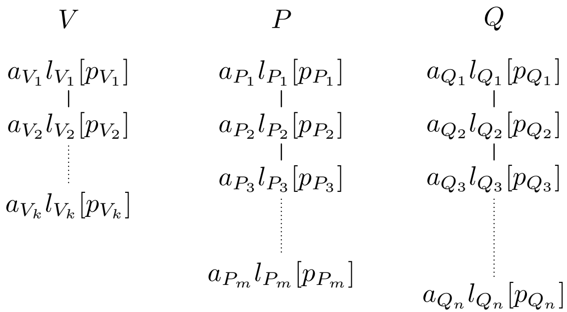

V
P
Q
$a_{V_1}l_{V_1}[p_{V_1}]$
$a_{V_2}l_{V_2}[p_{V_2}]$
$a_{V_k}l_{V_k}[p_{V_k}]$
$a_{P_1}l_{P_1}[p_{P_1}]$
$a_{P_2}l_{P_2}[p_{P_2}]$
$a_{P_3}l_{P_3}[p_{P_3}]$
$a_{P_m}l_{P_m}[p_{P_m}]$
$a_{Q_1}l_{Q_1}[p_{Q_1}]$
$a_{Q_2}l_{Q_2}[p_{Q_2}]$
$a_{Q_3}l_{Q_3}[p_{Q_3}]$
$a_{Q_n}l_{Q_n}[p_{Q_n}]$

Fig. 3. Representation of Queries

Based on the criteria for query/view answerability, for any V that answers P and Q, we have the followings:

$$\begin{aligned} V \vDash P \Rightarrow\ & a_{V_i} = a_{P_i} && 1 \le i \le k-1; \\ & l_{V_i} = l_{P_i} && 1 \le i \le k-1; \\ & p_{V_i} \equiv p_{P_i} && 1 \le i \le k-1; \\ & a_{P_k}l_{P_k}[p_{P_k}] \cdots a_{P_m}l_{P_m}[p_{P_m}] \sqsubseteq a_{V_k}l_{V_k}[p_{V_k}] \\ V \vDash Q \Rightarrow\ & a_{V_i} = a_{Q_i} && 1 \le i \le k-1; \\ & l_{V_i} = l_{Q_i} && 1 \le i \le k-1; \\ & p_{V_i} \equiv p_{Q_i} && 1 \le i \le k-1; \\ & a_{Q_k}l_{Q_k}[p_{Q_k}] \cdots a_{Q_n}l_{Q_n}[p_{P_n}] \sqsubseteq a_{V_k}l_{V_k}[p_{V_k}] \end{aligned}$$

With regards to the minimal query, the number of steps should be as long as possible (e.g. $a[./b/c] \vDash a/b[./c] \vDash a/b/c$), and the predicates should be as restrictive as possible. Therefore, for the minimal query V that answers both P and Q, k should be the first position where $s_{P_k} \not\equiv s_{Q_k}$, i.e., some of the following conditions are not satisfied: $a_{P_k} = a_{Q_k}$, $l_{P_k} = l_{Q_k}$ or $p_{P_k} \equiv p_{Q_k}$.

Next step is to find a predicate that minimally contains both predicates: $a_{P_k}l_{P_k}[p_{P_k}] \cdots a_{P_m}l_{P_m}[p_{P_m}]$ and $a_{Q_k}l_{Q_k}[p_{Q_k}] \cdots a_{Q_n}l_{Q_n}[p_{Q_n}]$. Logically, the minimal predicate containing two predicates that are defined on the same schema is the union of them, i.e., using "or". Above construction is unique, and any other query V' that answers both P and Q must answer V based on the query/view answerability, which proves that V is minimal. ■

We denote by $Q_1 \circ Q_2$ the minimal query that may answer both Q_1 and Q_2. Note that we say the minimal query is unique in terms of equivalence, e.g., a/b and $a[./b]/b$ are different in syntax but always produce the same result. Therefore, $Q_1 \circ Q_2$ is a singleton (i.e. only one view instead of a set of views). Furthermore, we have the following proposition.

Proposition 1. *If a query P answers a query Q, then the minimal query that answers P and Q is P, i.e., $P \circ Q = P$ iff $P \vDash Q$.*

Algorithm 1 MINQUERY(P, Q)

```
1:  $P = s_{P_1}\ s_{P_2} \cdots s_{P_m}$;
2:  $Q = s_{Q_1}\ s_{Q_2} \cdots s_{Q_n}$;
3:  if $s_{P_1} \not\equiv s_{Q_1}$ then
4:     $V = /*$, or $P$ if $Q \sqsubseteq P$, or $Q$ if $P \sqsubseteq Q$;
5:     return $V$;
6:  else
7:        $V = s_{P_1}$;
8:  end if
9:  $k = \mathsf{min}(m, n)$;
10: for $i$ from 2 to $k$
11:    if $s_{P_i} \equiv s_{Q_i}$
12:       $V += s_{P_i}$;
13:    else
14:       break
15:    end if
16: end for
17: $V += [s_{P_{i+1}} \cdots s_{P_m}$ or $s_{Q_{i+1}} \cdots s_{Q_n}]$;
18: return $V$
```

Computing the minimal query. We describe an algorithm to compute the operation $Q_1 \circ Q_2$. Recall that a query Q can be represented as a sequence of steps as: $Q = s_{Q_1}\ s_{Q_2} \cdots s_{Q_n}$, and each step is represented in the form: Axis NameTest Predicates? $(a_{Q_i}\ l_{Q_i}\ p_{Q_i})$. Two steps are *equivalent*, if their Axis, Name and Predicates? are equivalent, correspondingly. Algorithm 1 shows how to compute the minimal query. The correctness of this algorithm can be directly verified from the proof of Theorem 1 (by construction).

Next we illustrate why the joined view $V = /*$ if $s_{P_1} \not\equiv s_{Q_1}$ and P, Q are not contained by each other (lines 4-5). Here, we may safely omit some predicates. For example, given $P : a[./c/d/e]$ and $Q : b[./c/d/e]$, the minimal query $V = P \circ Q = /*$ while not $V' = /*[./c/d/e]$ which seems more restrictive. There are only two cases for predicates: `true` or `false`. (1) $[./c/d/e]$ is `true`, to materialize V' is equivalent to materialize V, the *root* element. (2) $[./c/d/e]$ is `false`, the result of V' is empty. We do not materialize a query with empty result. Both bases are normalized to $/*$.

We consider the following cases for minimizing the predicates. (1) Comparison predicates. The minimization of comparisons with numeric constant is straightforward. For example, $n < x$ `and` $n < y$ can be minimized to $n < x$ iff $x \leq y$; $n > x$ `or` $n > y$ can be minimized to $n > x$ iff $x \geq y$, etc. (2) Path minimization. We have P or $Q = Q$ if $P \sqsubseteq Q$; P and $Q = P$ if $P \sqsubseteq Q$. Algorithms to find the minimized query are applicable to our approach, which are omitted here due to space constraints. [12] and [1] investigate to minimize comparison predicates and minimize tree pattern queries, respectively.

3.3 Optimality of Candidate Views

In this section, we first describe the cost model, as a criterion for measuring views to answer a set of queries. We then discuss how to generate a view set as candidates, which is guaranteed to be a superset of the optimal solution and safely avoids enumerating all potential views.

We use SIZE$(\mathcal{X}, V)$ to denote the result size of applying view V over an XML database $\mathcal{X}$. When the XML database $\mathcal{X}$ is clear from the context, we use SIZE(V) as a simplification. Without loss of generality, we assume that the materialized views do not have index support. Evaluating a query over a materialized view requires one scan of the materialized XML fragments. We use CARD(V) to denote the number of labels in an XML view V. We have the following general cost estimation model of evaluating query Q based on view V materialized over an XML database $\mathcal{X}$:

$$\text{COST}(\mathcal{X}, V, Q) = \alpha \cdot \text{SIZE}(\mathcal{X}, V) + \beta \cdot \text{SIZE}(\mathcal{X}, V) \cdot \text{CARD}(Q) \qquad (2)$$

When the view is materialized on a disk, the overhead is dominated by disk `I/O`, in which case $\alpha \gg \beta$. Otherwise, if the views are materialized in a semantic cache, the overhead is dominated by the computational cost, which is determined by the materialized view fragments and the query size, thus $\alpha \ll \beta$.

Naturally, we have $\text{SIZE}(Q') \geq \text{SIZE}(Q)$ if $Q' \vDash Q$. For two views V and V', we say V is *better* than V' if V can be used to obtain a smaller value of Equation 1. Given a single query Q and two views V and V', where $V \vDash Q$ and $V' \vDash Q$, V is better than V' in answering Q iff $\text{SIZE}(V) < \text{SIZE}(V')$. Furthermore, given a query set $\mathcal{Q}$ and two views V and V', where $V \vDash \mathcal{Q}$ and $V' \vDash \mathcal{Q}$, V is better than V' in answering $\mathcal{Q}$ iff $\text{SIZE}(V) < \text{SIZE}(V')$. Recall that $V \vDash \mathcal{Q}$ means that V may answer each query Q in $\mathcal{Q}$.

There are other factors that might influence the cost model, which are not considered here. These factors may include the storage model of the materialized XML fragments, and different indices that may be exploited, which may lead to a more complicated model. However, we believe that our cost model, being simple but general, enables us to investigate representative algorithms.

Finding a candidate set. Next we discuss how to generate a candidate view set that is a superset of the optimal solution. We show some properties of the minimal query first. The minimal query, computed over the operator "$\circ$", satisfies the following identities:

(L1) $P \circ Q = Q \circ P$ (commutative law)
(L2) $Q \circ Q = Q$ (idempotent law)
(L3) $(P \circ Q) \circ V = P \circ (Q \circ V)$ (associative law)

The first two equivalences can be verified directly through Algorithm 1. We simply illustrate `L3` next. Based on Algorithm 1, assume that P, Q, V have k, m, n steps, respectively, and the ith step is the first step that s_{P_i}, s_{Q_i} and s_{V_i} are not equivalent. Both $(P \circ Q) \circ V$ and $P \circ (Q \circ V)$ are equivalent to the form: $s_{P_1} \cdots s_{P_{i-1}}[s_{P_i} \cdots s_{P_k}$ `or` $s_{Q_i} \cdots s_{Q_m}$ `or` $s_{V_i} \cdots s_{V_n}]$ such that identity `L3` holds. Thus, the computation of the minimal view of a query set is order independent.

Given a workload $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_n\}$, we first aggregate them pairwise to get $\binom{n}{2}$ views, which are obtained by computing $Q_1 \circ Q_2$, $Q_1 \circ Q_3$, etc. We then can aggregate $Q_1 \circ Q_2$ and $Q_1 \circ Q_3$ to get $Q_1 \circ Q_2 \circ Q_3$, and so on. We get $\mathcal{V} = \{Q_1, \cdots, Q_n, Q_1 \circ Q_2, Q_1 \circ Q_3, \cdots, Q_{n-1} \circ Q_n, \cdots, Q_1 \circ Q_2 \circ \cdots \circ Q_n\}$. There are $O(2^n)$ candidate views generated, which is deduced by $\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n - 1$. Although the worst case is still exponential, the size of candidate views is much smaller than all potential views. Furthermore, we will introduce a simple bound to significantly reduce the number of generated views.

Next we illustrate why we can safely ignore any view $V' \in \mathcal{C}$ ($\mathcal{C}$ is all views that can be generated) if $V' \notin \mathcal{V}$, while still ensuring optimality. This question comes from the fact that, for two queries Q_1, Q_2 and $V = Q_1 \circ Q_2$, there may exist another view V', where $V' \vDash V$ which is not considered for materialization. Here, V' can be potentially used to answer some query Q_3 but is not generated.

We prove this by showing that there exists another candidate view $V'' \in \mathcal{V}$ that is better than V'. Assume that $\mathcal{Q}'$ is a subset of $\mathcal{Q}$ that V' can answer each query in $\mathcal{Q}'$, i.e., $\mathcal{Q}' = \{Q | Q \in \mathcal{V} \wedge V' \vDash Q\}$. V' is not in $\mathcal{V}$, therefore, V' is not the minimal view to answer $\mathcal{Q}'$. Suppose that the minimal view to answer $\mathcal{Q}'$ is V''. Naturally, we have $V' \vDash V''$ and V'' is better than V' for the given workload $\mathcal{Q}$ to minimize Equation 1.

4 Candidate View Organization

View organization. We organize the candidate views $\mathcal{V}$ as a directed acyclic graph (named VCUBE), which is denoted as $\mathcal{G}(N, E)$. Each node $u \in N(\mathcal{G})$ represents a view u_V in $\mathcal{V}$. Each node u has a level, denoted as $\textsc{level}(u)$, which is the number of queries in the original workload $\mathcal{Q}$ that are used to generate the view u_V. There is an edge $(u, v) \in E(\mathcal{G})$, iff $\textsc{level}(u) = \textsc{level}(v) + 1$ and v_V can be used to generate u_V, i.e., the query set used to generate v_V is a subset of the query set used to generate u_V.

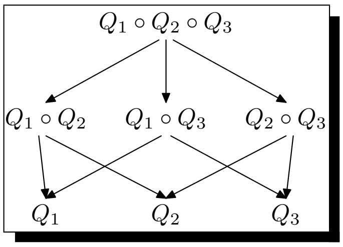


Fig. 4. Sample VCUBE

Example. Given a workload $\mathcal{Q}_e = \{Q_1, Q_2, Q_3\}$, the level of $Q_1 \circ Q_2$ is 2, i.e., $\textsc{level}(Q_1 \circ Q_2) = 2$. There is an edge from $Q_1 \circ Q_2 \circ Q_3$ to $Q_1 \circ Q_2$, since $\textsc{level}(Q_1 \circ Q_2 \circ Q_3) = 3$, $\textsc{level}(Q_1 \circ Q_2) = 2$ and $\{Q_1, Q_2\}$ is a subset of $\{Q_1, Q_2, Q_3\}$. The VCUBE for workload $\mathcal{Q}_e$ is shown in Figure 4.

It deserves noting that we only maintain edges between adjacent levels of views e.g. no edge from $Q_1 \circ Q_2 \circ Q_3$ to Q_1, although the former answers the latter. The nice property is that, the answerability is traced not only by direct edges, but also by reachability relationships. Based on VCUBE, our problem is reduced to finding a minimum weight set of nodes that covers all leaves and minimizes Equation 1. We say VCUBE guarantees optimality since the optimal view set is subsumed in all candidates and answerability is verifiable. Next we illustrate why we could safely avoid checking the answerability between two unrelated views e.g. $Q_1 \circ Q_2$ and Q_3.

In Figure 4, there are two edges from $Q_1 \circ Q_2$, to Q_1 and Q_2, respectively. Assume that $Q_1 \circ Q_2$ can also answer Q_3 but there is no edge from $Q_1 \circ Q_2$ to Q_3, whether we would be underestimating $Q_1 \circ Q_2$, since we consider it to answer Q_1 and Q_2 only. According to Proposition 1, $Q_1 \circ Q_2 \circ Q_3 = Q_1 \circ Q_2$ if $Q_1 \circ Q_2 \vDash Q_3$, and there should have existed a path from $Q_1 \circ Q_2 \circ Q_3$ to Q_3. Therefore, the optimal solution can always be identified. More specifically, assume that there are two nodes $u, v \in V(\mathcal{G})$ where $u_V \vDash v_V$ but there is no path from u to v. The nearest common ancestor (node w) of u and v satisfies $w_V = u_V \circ v_V = u_V$ since $u_V \vDash v_V$. The path from u to v could thus be omitted.

The benefits of edge construction in VCUBE are twofold. (1) The edges can be directly constructed when computing the minimal views, without checking the answerability between views, which is expensive. (2) The number of edges maintained is much smaller than the edges computed depending on answerability. This can reduce both the graph size (the number of edges) and the overhead of searching optimal solution over it.

View graph optimization. We describe to optimize the view graph by safely omitting some views to be generated. We introduce a special view as an upper bound. The special view is $\mathit{\Delta} : /*$, whose result is just the *root* element of $\mathcal{X}$ and would never be materialized. For any query Q, we have $Q \circ \mathit{\Delta} = \mathit{\Delta}$. Therefore, if some node in the bottom-up construction is $\mathit{\Delta}$, we do not generate any other nodes that may reach it, which can greatly reduce the candidates to be generated. Recall that in Equation 1, we have a size constraint B. Therefore, if some view V has the estimated materialized size larger than B, we do not materialized V and can safely set V to $\mathit{\Delta}$.

5 Materialized View Selection Algorithms

Finding the optimal solution for Equation 1 is NP-hard, which can be reduced to the set cover problem[1]. In this section, we describe heuristic methods to identify approximate solutions.

Estimating view size. View selection algorithms require knowledge of the size of each view. There are many ways of estimating the sizes of views [19, 23]. The approach we adopt is sampling, i.e., running views on a representative portion of the base data and multiplied by a scaling factor.

We have to consider space-time tradeoffs in view selection. In the space-optimized perspective, we would like to select the views with the smallest size to answer as many queries as possible. In the time-optimized perspective, we will materialize the query workload only, which can answer any given query directly but with a large size. We will first discuss an algorithm targeting space-optimized, followed by an algorithm for space-time consideration.

5.1 Space-Optimized Algorithm

The goal of space-optimized is to material the smallest XML fragments to answer a given query workload. We adopt a bottom-up, level-by-level dynamic programming strategy over the VCUBE.

Given a candidate view set $\mathcal{V}$ generated from a workload $\mathcal{Q}$, its VCUBE $\mathcal{G}(N, E)$ and a node $u \in N(\mathcal{G})$, we use $\textsc{views}(u)$ to represent the set of queries that generate the view u_V e.g. $\textsc{views}(Q_1 \circ Q_2) = \{Q_1, Q_2\}$. Conversely, for a subset of queries $\mathcal{Q}'$, we denote by $\textsc{genv}(\mathcal{Q}')$ the views generated by aggregating each query in $\mathcal{Q}'$ e.g. $\textsc{genv}(Q_1, Q_2) = Q_1 \circ Q_2$. Our recursive definition of the minimum cost of computing $\textsc{size}(u_V)$ for each node $u \in N(\mathcal{G})$ is as follows:

$$\textsc{size}(u_V) = \textsc{min}\{\textsc{size}(\textsc{genv}(\textsc{views}(u_V) - \textsc{views}(v_V))) + \textsc{size}(v_V), \textsc{size}(u_V)\} \quad (3)$$

here, node $v \in N(\mathcal{G})$ is any graph node that is reachable from u. For instance, when computing the size of $Q_1 \circ Q_2 \circ Q_3$, we compare its size with the sum of sizes of each combination in $(Q_1,\ Q_2 \circ Q_3)$, $(Q_2,\ Q_1 \circ Q_3)$ and $(Q_3,\ Q_1 \circ Q_2)$, and record the smallest one.

[1] http://en.wikipedia.org/wiki/Set_cover_problem

```
Algorithm 2 SpaceOptimal($\mathcal{G}$)
1: for $i$ from 1 to $|N(\mathcal{G})|$ do
2:    estimate size($n_i$)
3: end for
4: for $i$ from $|\mathcal{Q}| + 1$ to $|N(\mathcal{G})|$ do
5:    for $j$ from 1 to $|N(\mathcal{G})|$ do
6:       if genv(views($n_i$) $\cup$ views($n_j$)) doesn't exist
            or (views($n_k$) = views($n_i$) $\cup$ views($n_j$)
            and size($n_k$) > size($n_i$) + size($n_j$))
7:          size($n_k$) $\leftarrow$ $size(n_i) + size(n_j)$
8:       end if
9:    end for
10: end for
```

The intuition of Equation 3 is to compute the smallest size for each graph node from all the views that it might answer. Algorithm 2 shows a dynamic programming based approach with time complexity $O(n^2)$ where n is the number of graph nodes. Here, we use n_i to represent the i-th node in the bottom-up, level-by-level traversal mode. Initially, each view has an estimated size (lines 1-3). In each iteration (leaf views could be skipped in line 4), we compute the size of one view V by counting all the views that are answerable by V, but marking the smallest size only. Note that a view might not exist in graph originally, if its size exceeds a given upper bound, while this view will be used (not materialized) in selecting views in our dynamic program. After computing the last view (the graph root), we could find the smallest XML fragments to materialize to answer the given query workload $\mathcal{Q}$. We illustrate this algorithm by an example next.

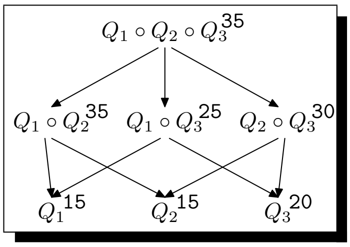


Fig. 5. vcube with Sizes

Example. Consider the vcube in Figure 5, each view is associated with an estimated size. Assume that the size constraint is $B = 40$. We start with an initial view set $\mathcal{V} = \{Q_1, Q_2, Q_3\}$. Consider $Q_1 \circ Q_2$, which is omitted since size$(Q_1 \circ Q_2) = 35 >$ size$(Q_1)+$size$(Q_2) = 30$. However, $Q_1 \circ Q_3$ and $Q_2 \circ Q_3$ are considered to be materialized since size$(Q_1 \circ Q_3) <$ size$(Q_1)+$size(Q_3) and size$(Q_2 \circ Q_3) <$ size$(Q_2)+$size(Q_3). To answer all queries $\mathcal{Q}$, materializing $Q_1 \circ Q_3$ and Q_2 requires the size 40, while for $Q_2 \circ Q_3$ and Q_1, it requires $45 > 40$. Therefore, we replace Q_1, Q_3 in $\mathcal{V}$ by $Q_1 \circ Q_3$ and $\mathcal{V} = \{Q_1 \circ Q_3, Q_2\}$.

Next we consider $Q_1 \circ Q_2 \circ Q_3$, whose estimated size is less than the summation of $Q_1 \circ Q_3$ and Q_2, i.e., size$(Q_1 \circ Q_2 \circ Q_3) = 35 <$ size$(Q_1 \circ Q_3)+$size$(Q_2) = 40$. Furthermore, size$(Q_1 \circ Q_2 \circ Q_3) = 35 < B = 40$, thus we have $\mathcal{V} = \{Q_1 \circ Q_2 \circ Q_3\}$.

5.2 Space-Time Algorithm: A Greedy Solution

In above example, the space-optimized approach only considers to use the smallest sized views to answer the given workload. However, materializing $Q_1 \circ Q_3$ and Q_2, whose total size is 40, requires a larger size but may have a lower query execution time.

Next we consider a space-time optimized approach. We define a utility function on the view graph, for each candidate view V and a workload $\mathcal{Q}$, where each query $Q_i \in \mathcal{Q}$ is associated with a weight w_i.

$$\textsc{util}(V) = \frac{\Sigma_{V \models Q_i} w_i}{\textsc{size}(V)} \tag{4}$$

here, we compute the utility of a view V by considering how it can be used to improve the query response time. The utility value is in inverse proportion to the size of view, which means that the smaller the materialized size of a view, the larger utility value it has. The utility value is in direct proportion to the summated weights of queries the view may answer, which means that the more queries the view can answer, the larger the utility value will be. The weight of a view could be query independent as query frequency or query dependant like *full-resp-time/view-resp-time*.

Algorithm description. We simply describe the algorithm of heuristically selecting a set of views. (1) Compute the utility value of each view in the view graph, and select the one with the largest utility value. (2) Remove the selected view and all queries it may answer. Recompute the utility values of remaining views. (3) Repeat this procedure until all queries can be answered or the total size exceeds the size constraint B. This greedy strategy is similar to the one used in [11] for computing data cube.

6 Performance Study

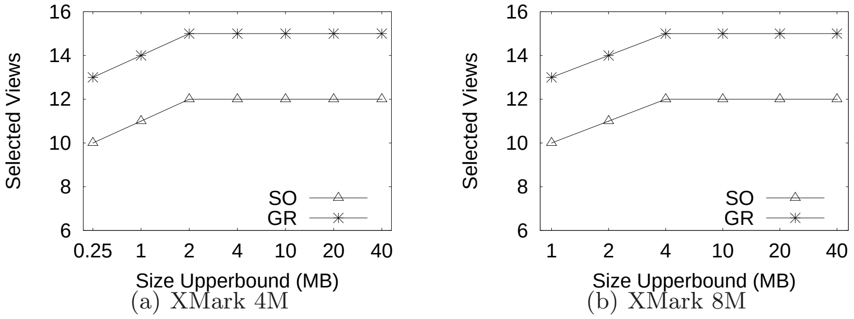
We report on empirical results in two orientations. Firstly, we measure the two algorithms for selecting materialized views. Secondly, we study the performance gain using materialized views, against the evaluation over the base data (BD for short). For simplicity, we represent our space-optimized algorithm as SO, and the greedy strategy as GR.

The experiments were conducted on a PC 1.6GHz machine with 512MB of memory, running Windows XP. The algorithms were coded in C++. We used Berkeley DB XML[2] for managing XML data with basic index support and storing views. We used XMark as our test benchmark, with 16 queries (not listed for space consideration). Q_{1-4} are in the form /site/people/person[@id="person#"]//? where person# represents a person id (e.g. person10) and ? a NameTest (e.g., name); queries Q_{5-8} are in the form /site/regions//item[@id="item#"]//?; queries Q_{9-12} are in the form /site/closed_auctions/closed_auction//?; and queries Q_{13-16} are in the form //open_auctions//open_auction[Expr1][Expr2]//? where Expr1 is a path predicate.

6.1 Selecting Views in Practice

We mainly measure the effect of size parameters in two selection algorithms. The first group of experiments is to fix the size limitation for each single view large enough (e.g. the document size), while varying the total size upper bound (B). Figure 6 shows the number the views selected when varying the upper bound B

[2] http://www.oracle.com/database/berkeley-db/xml/index.html



(a) XMark 4M (b) XMark 8M

Fig. 6. Varying Size Upperbound B

for different documents. Here in x-axis, 0.25 means 0.25MB and 1 for 1MB, etc. Both sub-figures show that, along with the enlarging size constraint, the number of views for both algorithms increase. For example, for 4MB document, the SO selects 10 to 12 views and GR picks 13 to 15 views. When the constraint B exceeds some threshold (e.g. 2MB for 4MB document), the size constraint cannot affect the result of view selection. This group of experiments also tells that we could use a relative small size constraint, e.g., half of document size for XMark, to get the optimal solution. The benefit to select a small size bound is that, answering queries over base data could be accelerated by underlying indices. This could be faster than a materialized view without index support.

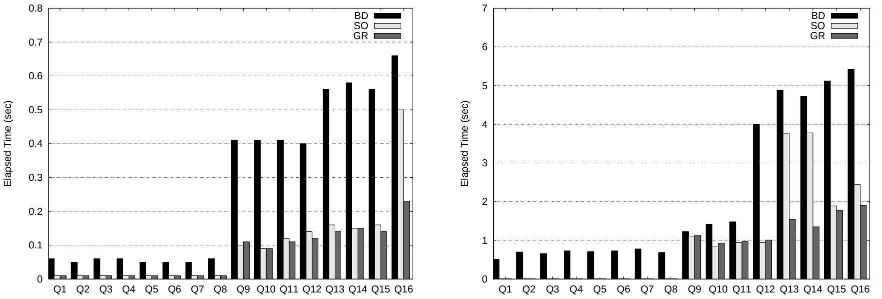
We also examine how the size limit of each view (parameter b) affects the size of VCUBE in terms of number of graph nodes. Recall that in a VCUBE, if the size of some view exceeds b, we don't generate this graph node. Theoretically, the VCUBE size of a workload of 16 queries is $2^{16} = 65536$. Take an Xmark 4MB document and fixed B =1M. If we set b to be 4MB, there are 24 nodes in VCUBE. When we change b to 1MB, there are 16 nodes generated. The cause of this result comes from two facts: (1) the views that might answer queries in different groups will have a large size and thus are not materialized; (2) the number of views materialized for the same group of queries decreases when we restrict b.

Due to the small size of VCUBE, the costs for both selection strategies are surprisedly small. Note that the size of each view is pre-estimated using a small sample. In our case, we use an XMark 1MB document and estimate with a scaling factor (e.g. 4 for an XMark 4MB document). Therefore, the selection is only affected by the number of graph nodes but not the document size. It takes 10ms for SO strategy and 50ms for GR strategy, and this basically keeps the same for different sized XMark documents.

In the worst case, the estimated view size via aggregating two queries exceeds b. This case is reflected in above tests where no views generated for two groups of queries. However, the query might be materialized if its size is below b.

6.2 Answering Queries using Materialized Views

We compare the response time of SO, GR and BD (without using views), on top of Berkeley DB XML. We test XMark documents for different factors, from



(a) XMark 4M (b) XMark 32M

Fig. 7. Answering Queries using Views

1, 2 to 32. The weights of queries are all set to 1 as a normalized frequency in equivalent weights. All tests give similar results, so we only report two documents in Figure 7, where x-axis carries all test queries and y-axis its response time in millisecond. This group of experiments verifies two important goals as expected: (1) Answering queries using materialized views is much faster than evaluating over the base data, even if the base data is well indexed. The reason is simple, as the materialized views have smaller size and many structural joins of queries have been pre-evaluated by views. (2) GR outperforms SO in terms of response time. This comes from the different aims of algorithm designs. SO aims at an optimized space consumption, while GR balances the space overhead and cost estimation model for query processing, which is simple yet general and effective.

7 Conclusion and Future Work

We have described a framework to select XML views for materialization. For a given workload, we present a new way to generate a small number of candidates while ensuring optimality. Based on a well organized graph structure (VCUBE) for maintaining a minimal set of candidate views, we present two heuristic algorithms for view selection. We experimentally demonstrate that query response time could be significant reduced using materialized views. Moreover, the VCUBE gives full possibilities to develop other algorithms with different aims and for further optimization.

In the future: (*i*) We plan to further develop algorithms over VCUBE, for both better selected views and faster computation; (*ii*) In this paper, we store materialized views using a disk-based database. We would like to cache views using main-memory databases, to examine the advantage of selecting materialized views in different environments. (*iii*) We want to implement view selection as a transparent optimization strategy, which is self-tuning and works off-line by analyzing query logs. (*iv*) Incremental (or lazy) materialized view maintenance is an interesting topic, compared with re-materialization each time for changed base data and query logs. (*v*) We also plan to investigate, as a relaxed version, the problem of view selection that allows to access the base data.

References

1. S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.
2. S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and content scoring for XML. In *VLDB*, 2005.
3. S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible structure and full-text querying for XML. In *SIGMOD*, 2004.
4. A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.
5. E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, 1997.
6. R. Chirkova, A. Y. Halevy, and D. Suciu. A formal perspective on the view selection problem. In *VLDB*, 2001.
7. W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular xpath queries on xml views. In *ICDE*, 2007.
8. S. D. Gribble, A. Y. Halevy, Z. G. Ives, M. Rodrig, and D. Suciu. What can database do for peer-to-peer? In *WebDB*, 2001.
9. H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, 1997.
10. H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, 1999.
11. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, 1996.
12. J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM J. Comput.*, 2(4), 1973.
13. H. J. Karloff and M. Mihail. On the complexity of the view-selection problem. In *PODS*, 1999.
14. D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4), 2000.
15. L. V. S. Lakshmanan, H. Wang, and Z. J. Zhao. Answering tree pattern queries using views. In *VLDB*, 2006.
16. B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.
17. G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.
18. I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD Conference*, 1997.
19. N. Polyzotis, M. N. Garofalakis, and Y. E. Ioannidis. Selectivity estimation for xml twigs. In *ICDE*, 2004.
20. A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. E. Abbadi, and K. S. Candan. Maintaining XPath views in loosely coupled systems. In *VLDB*, 2006.
21. A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path expression views. In *SIGMOD Conference*, 2005.
22. N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, 2008.
23. W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom histogram: Path selectivity estimation for xml data with updates. In *VLDB*, 2004.
24. W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.
25. J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, 1997.