RC25148 (W1104-066) April 15, 2011
Computer Science

IBM Research Report

Optimizing Access across Multiple Hierarchies in DataWarehouses

Lipyeow Lim
University of Hawaii at Manoa

Bishwaranjan Bhattacharjee
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Optimizing Access Across Multiple Hierarchies in Data Warehouses

Lipyeow Lim
University of Hawai'i at Mānoa
lipyeow@hawaii.edu

Bishwaranjan Bhattacharjee
IBM T. J. Watson Research Center
bhatta@us.ibm.com

Abstract

In enterprise data warehouses, different users in different business units often define their own application specific dimension hierarchies tailor made to their reporting and business performance monitoring needs. Due to resource constraints, only on a small number of these hierarchies are precomputed for performance optimization. Consequently aggregations over hierarchies without precomputations are often less responsive. We report on a performance problem in a very large banking enterprise where the large number of application specific hierarchies became a performance bottleneck. This paper proposes a novel solution for optimizing the performance of data warehouses with a large number of application specific hierarchies. We exploit the observation that dimension hierarchies in real data warehouses often contain significant overlaps. Our method detects common sub-structures among hierarchies and provides a rewriting algorithm to exploit any precomputations on these shared sub-structures. Our solution is applicable to data warehouses of large enterprises with a large number of business units and hence a large number of application specific hierarchies.

1. Introduction

Data warehouses and on-line analytical processing (OLAP) have gained widespread use in almost all parts of an enterprise for decision support and business performance monitoring. A typical deployment uses the two-tier data warehousing approach [5] as illustrated in Figure 1. The data repository tier consists of one or more DBMS (eg. IBM DB2 UDB) that extracts, transforms and cleans the data from multiple sources. The data access tier consists of one or more data marts (eg. Hyperion Essbase, Cognos etc) through which users access subsets of the data for subject-specific OLAP. An OLAP data mart consists of a fact table, which can be thought of as a materialized view of the data in the data repository, and several dimensions, each of which can

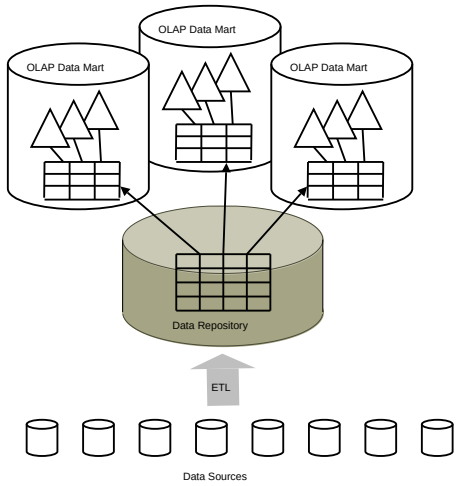
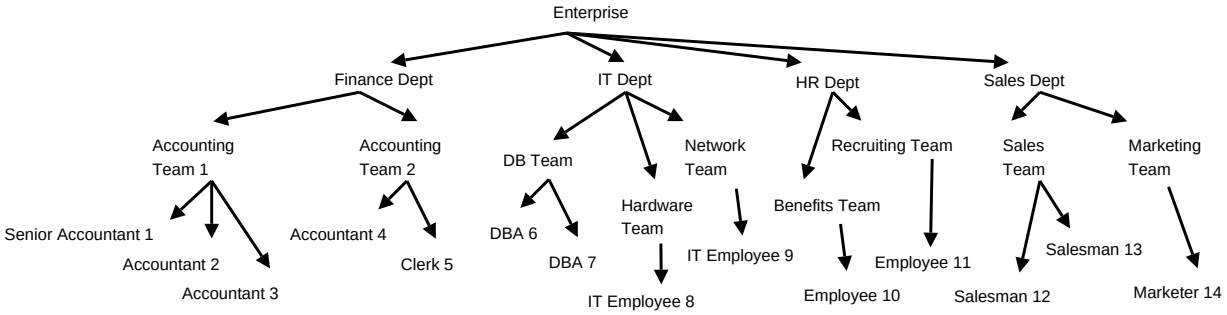


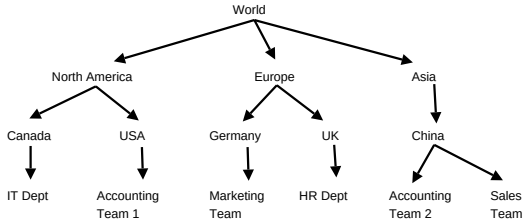
Figure 1: OLAP deployments in the industry typically uses the two-tier data warehousing architecture consisting of a data repository and multiple OLAP data marts. A triangle denotes a hierarchy.

be associated with multiple complex and unbalanced hierarchies. Different users in the enterprise typically define their own application-specific hierarchies within the same data mart. For example, within the same OLAP data mart for sales transactions, an accountant might define a hierarchy for aggregating sales transactions across all the business units in the enterprise, and a marketing executive might define another hierarchy to aggregate sales transactions by geographical regions. In many real usecases, a set of enterprise-wide primary hierarchies are defined on each common fact table. Different users, lines of business, and business units then define their own application-specific hierarchies such that the leaf nodes of each application-specific hierarchy point back to elements in the primary hierarchies. Figure 2 illustrates an example of a primary hierarchy and two application specific hierarchies defined on it.

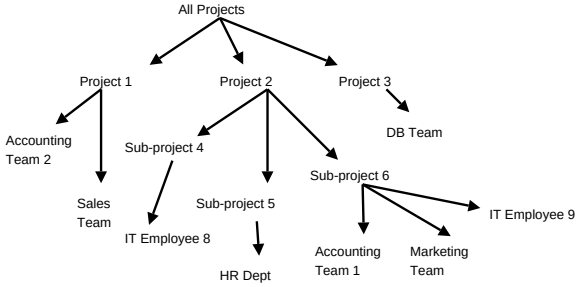
In the OLAP environment, user queries are defined



(a) Primary hierarchy for the organization dimension.



(b) Application-specific geography hierarchy



(c) Application-specific project hierarchy

TransactionID	EmployeeID	Time	ExpenseType	Amount
10001	DBA 1	2007-05-01-2059	Harddisk	1000
10002	Accountant 2	2007-05-03-0759	Stationary	100
10003	DBA 1	2007-05-03-0859	Software	2000
...	...	...	...	...

(d) The expense transactions fact table.

Figure 2: Primary organization hierarchy and hierarchies specific to expense accounts applications. The geographical hierarchy supports aggregations of expenses by geographical regions or the business units. The project hierarchy supports aggregations of expenses according to projects. Note that the leaf nodes of the application-specific hierarchies point to nodes in the primary hierarchy.

against a small number of application-specific hierarchies. These queries are then translated into an equivalent query on the fact table automatically. Query optimization usually involves precomputing certain aggregates on some hierarchies.

Example 1 (Precomputing aggregates) *Consider a query that aggregates the expenses in the fact table (Figure 2(d)) according to the geographical hierarchy in Figure 2(b) and reports the expenses by continent (North America, Europe, and Asia). To speedup the query, the administrator can specify that expense aggregates for certain countries or certain continents be precomputed and stored in the OLAP environment, so that the query optimizer can exploit them.*

If the data warehouse has unlimited resources, every aggregation on every hierarchy can be precomputed in materialized views. With limited resources, only a small subset of aggregations can be precomputed.

In small and medium enterprises, the number of application-specific hierarchies tend to be small in the order of tens; hence, current data warehouse optimization techniques such as materializing summary tables are adequate. In large enterprises, there could hundreds of business units or lines of business, resulting in hundreds of application-specific hierarchies. Precomputing aggregates in summary tables for every internal node of every application-specific hierarchy is not feasible due to large space requirement. Moreover, the OLAP ad-

ministrator typically creates precomputed aggregates for a few important hierarchies. Can we optimize access on hierarchies on which no precomputed aggregations have been created ? Given that there are overlaps in the application-specific hierarchies, such optimization should be possible in principle. If precomputed aggregates have been created on a hierarchy H_1 and H_1 overlaps with hierarchy H_2, a query written on hierarchy H_2 should be able to exploit the precomputed aggregates on the overlapping portion of H_1 and H_2.

Example 2 (Exploiting overlapping hierarchies) *Consider the same query in Example 1 that reports the total expenses by continent (North America, Europe, and Asia) according to the geographical hierarchy in Figure 2(b). Further suppose no precomputed aggregates exists for the geographical hierarchy, but precomputed aggregates exists for the projects in the project hierarchy of Figure 2(c). The query should be able to exploit the precomputed aggregates for the "Project 1" node of the project hierarchy, because the aggregate is equivalent to the "Asia" or "China" node in the geographical hierarchy.*

Unfortunately, current OLAP environments are not aware of overlaps between hierarchies and hence are not able to rewrite queries to exploit precomputed aggregates across different hierarchies.

The failure to exploit hierarchy overlap information resulted in a serious performance problem in a large banking enterprise that we had the opportunity to work with. The banking enterprise had a financial datamart (chart of accounts) which used a commercially available OLAP engine as a front end and a relational engine as the backend. The datamart design allows for dimensions for accounts, companies and cost centers. The key problem is that the financial company has 200,000 cost centers and each of them has the ability to define its own organizational hierarchies. This led to the dimension table having more than 100 million rows. It has resulted in serious performance problems with only 10 to 20 users being able to use the system simultaneously from a pool of 1500 users. It was determined that if common sub-trees were identified from the hierarchies and precalculated in the fact table, it would allow system performance to improve dramatically.

Our contributions. In this paper, we propose a method for optimizing aggregation queries on a set of hierarchies using precomputed aggregates from another set of hierarchies. Our method exploits the fact that most hierarchies in an industrial setting contain a significant amount of overlap and that these overlaps result in equivalences that can be used in query rewriting. Our method consists of two phases. In the off-line phase, we propose an algorithm to discover overlapping relationships in the hierarchies of the OLAP environment. These relationships are then stored in the catalog tables of the OLAP server. In the on-line phase, we propose a query rewrite algorithm that leverages the overlapping relationships discovered in the off-line phase to rewrite the OLAP queries. In summary, the contributions of this paper are:

- We highlight an important and practical problem when data warehousing and OLAP are deployed in the industry. To the best of our knowledge, this problem has not been adequately addressed in previous literature.
- We propose an algorithm to discover and store overlap relationships in the hierarchies of an overlap environment.
- We propose an algorithm to rewrite queries using the discovered relationships, so that precomputed aggregates can be better exploited.
- Our method addresses a limitation of current OLAP environments to exploit precomputed aggregates across different hierarchies.
- We prototyped our method and evaluated empirically the effectiveness of the proposed method.

Paper organization. Section 2 describes our method in detail. Section 3 presents the experimental evaluation. Section 4 discusses related work. We conclude in Section 5.

2. Our Approach

2.1. Overview

Our approach to optimize aggregation queries on a set of hierarchies using precomputed aggregates from another set of hierarchies consists of two phases.

The off-line phase scans all the application-specific hierarchies in the OLAP environment in order to discover equivalences or overlaps between each pair of hierarchies. Naturally, only application-specific hierarchies of the same dimension can have equivalences, so comparisons of application hierarchies across different dimensions are not necessary. Section 2.2 describes the algorithms for discovering these equivalences or overlaps in greater detail. The result of the off-line phase is a list of node pairs for each pair of hierarchy. A node pair (u, v) for hierarchy pair (i, j) denotes that the node

identifier (node ID) u from hierarchy i is equivalent to node ID v from hierarchy j in the sense that an aggregation based on the sub-tree rooted at u will yield the same result as an aggregation based on the sub-tree rooted at v. These equivalence pairs can be stored in a catalog table with the following schema,

```
Overlap( DimensionID, PrimaryHierarchyID,
         HierarchyID_A, HierarchyID_B,
         NodeID_A, NodeID_B ).
```

Our algorithms rely on two assumptions: (1) siblings in each hierarchy is ordered using some canonical ordering such as recursively ordering each sub-tree according to their leftmost leaf labels, and (2) the Dewey labeling scheme [18] is assumed for node identification. Node IDs can therefore be implemented as Dewey identifiers. While our algorithms only require the Dewey IDs to be unique within each hierarchy, Dewey IDs that are unique across the entire data mart can also be used.

The on-line phase rewrites a given query using the equivalence pairs from the off-line phase. An OLAP query is typically generated by a graphical user interface based on an application-specific hierarchy. The particular type of query we are concerned with are aggregations of a fact table column according to all the leaf labels of some sub-tree of the application-specific hierarchy. Note that queries consisting of more complex expressions can often be reduced to several of such simple aggregation sub-queries. Our rewriting algorithm proceeds as follows. The leaf nodes associated with the query are merged according to the associated hierarchy into sub-trees. A greedy set cover algorithm is used to find the set of sub-trees that cover the query. We then check the `Overlap` table from the off-line phase to find equivalent sub-trees in other hierarchies. Moreover, we only want equivalent sub-trees whose results have been materialized or cached in the OLAP environment. The output of the on-line phase consists of a list of equivalent sub-trees for each sub-tree associated with the query. If each list contains more than one sub-tree, the optimizer can pick one using heuristics or cost-based metrics.

2.2. Off-line Phase: Discovering Overlaps

The purpose of the off-line phase is to discover equivalences or overlaps in the application-specific hierarchies for each dimension. Without loss of generality, the rest of the section describes the off-line phase assuming hierarchies from a single dimension taking a set of values D. To extend to more than one dimension, the same algorithms can be run on each dimension independently. Before we describe the algorithmic details of our off-line phase, we define hierarchies and the notion of overlap more formally.

Definition 1 *A hierarchy h is a quadruple (V, E, l, L) where V is a set of nodes, E is a set of edges, $l : V \mapsto L$ is a function that maps nodes to labels in the label set L, and (V, E) is a tree.*

In addition, we will use $\mathcal{L}(h)$ to denote the set of leaf nodes of h, and $Adj(v)$ to denote the set of children of the node v.

Definition 2 *A primary hierarchy $p = (V, E, l, L)$ for a dimension with values D is a hierarchy where $\forall u \in \mathcal{L}(p), l(u) \subseteq D$*

Figure 2(a) illustrates an example of a primary hierarchy for the organization dimension.

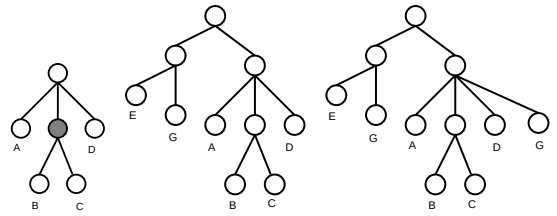
Definition 3 *An application-specific hierarchy h on a primary hierarchy $p = (V_p, E_p, l_p, L_p)$ is a hierarchy where $\forall u \in \mathcal{L}(h), l(u) \in \{l(v) : \forall v \in V_p\}$*

Figure 2(b) illustrates an example of an application-specific hierarchy based on the organization primary hierarchy. Another important characteristic of the hierarchies that we consider in this paper is that if we consider the dimension values that a hierarchy (whether primary or application-specific) covers, each dimension value should not be covered more than once in any hierarchy. If this condition does not hold, the normalization techniques from Treescape [16, 15, 16] can be used to transform the hierarchies into well-behaving ones. We now formalize the intuition of two hierarchies sharing a sub-hierarchy.

Definition 4 *A hierarchy $g = (V_g, E_g, l_g, L_g)$ is a matching sub-hierarchy of an application-specific hierarchy $h = (V_h, E_h, l_h, L_h)$ denoted by $g \preceq h$ iff there is a bijection f between V_g and a subset $V'_h \subseteq V_h$, such that*

1. *for all $(u, v) \in E_g$, $(f(u), f(v)) \in E_h$, and*
2. *for all $u \in V_g$, $Adj(f(u)) = \{f(v) : \forall v \in Adj(u)\}$, and*
3. *for all $u \in \mathcal{L}(h)$, $l_g(u) = l_h(f(u))$.*

The first condition ensures that the tree (V_g, E_g) associated with hierarchy g is isomorphic to the sub-tree induced by the node set V'_h. The second condition ensures that the bijection f maps g to an entire sub-tree of h, i.e., all descendant nodes of the node set V'_h are included in V'_h. The third condition ensures that the labels



(a) (b) (c)

Figure 3: Examples of matching and non-matching sub-hierarches. Figure 3(a) is a matching sub-hierarchy of Figure 3(b) but not a matching sub-hierarchy of Figure 3(c).

of the leave nodes in the bijection need to match; however, labels of internal nodes need not match. Figure 3 illustrates an example of matching sub-hierarchies. Figure 3(a) is a matching sub-hierarchy of Figure 3(b) but not a matching sub-hierarchy of Figure 3(c) because the second condition has been not be satisfied.

Definition 5 *Two application-specific hierarchies h_1 and h_2 on the same primary hierarchy p has a common matching sub-hierarchy h iff h is a matching sub-hierarchy of h_1 and h is also a matching sub-hierarchy of h_2.*

The notion of a common matching sub-hierarchy is to formalize the intuition for an *overlap* or *overlapping region* between two hierarchies.

Definition 6 *A common matching sub-hierarchy h of two application-specific hierarchies h_1 and h_2 is maximal if there does not exists a common matching sub-hierarchy h' with a larger node set such that h is a matching sub-hierarchy of h'.*

In Figure 3, the sub-hierarchy rooted at the shaded node is a maximal common matching sub-hierarchy of Figure 3(b) and Figure 3(c), while the single node sub-hierarchy rooted at the node with label 'B' is a common matching sub-hierarchy but not a maximal common matching sub-hierarchy.

In general two application-specific hierarchies can have several maximal common matching sub-hierarchies or overlaps. The goal of the off-line phase is to find all the maximal overlaps between all pairs of application-specific hierarchies for each dimension in the data mart. Algorithm 1 outlines the pseudo-code for the off-line phase.

The off-line phase first constructs an inverted index that maps a label (from the primary hierarchy) to a list

Algorithm 1 FINDOVERLAP(p, H)

Input: A primary hierarchy p, and a set of application-specific hierarchy H

Output: $\mathcal{M}(i,j)$, $\forall(i,j) \in H \times H$, is a set of node ID pairs of the root nodes of the maximal overlapping sub-hierarchies

```
1: Let L_p be the set if labels in the primary hierarchy p.
2: Scan through H and construct an inverted index I that
   maps a label from L_p to a list of hierarchy IDs from H
   that contain that label.
3: Sort I according to the size of the hierarchy ID list
4: Discard labels in I that occur in only one hierarchy
5: for all hierarchy ID list H_idx starting from the smallest
   list do
6:     for all hierarchy pair (i, j) in H_idx × H_idx do
7:         if done(i, j) is true then
8:             next;
9:         done(i, j) ←true
10:        C ← LeafLabels(i) ∩ LeafLabels(j)
11:        M(i, j) ←MERGEOVERLAP (C, i, j)
```

of application-specific hierarchies that contain that label in its leaf nodes – all the hierarchies in that list contain at least one leaf node with the same label. Line 3 sorts these inverted lists according to the size of each list. Line 4 prunes away inverted lists that contain only one hierarchy, because we are only interested in labels shared by at least two hierarchies. The sorted inverted lists provide an order for subsequent processing. The for-loop in line 5 iterates over each of these lists starting from the smallest list. Within each list, all pairs of hierarchies are compared. Lines 7-9 ensures that each pair of hierarchy (regardless of which inverted lists they belong to) is compared only once. The comparison begins by finding the set of common leaf labels of the two hierarchies in question. The function $LeafLabels(h)$ return the set of labels associated with the leaf nodes of hierarchy h. The bulk of the comparison work is then performed by the procedure MERGEOVERLAP as outlined in Algorithm 2.

The MERGEOVERLAP procedure takes as input two hierarchies i and j, and the set of common leaf labels. The procedure first finds the pairs of leaf nodes from each hierarchy that are associated with the common leaf labels and stores them in M. These pairs of leaf nodes are then sorted according to the Dewey IDs of the first node in each pair. The sorted pairs in M are then repeated scanned to merge groups of sibling nodes into sub-trees. Since we are dealing with pairs of nodes from two hierarchies (i and j), the merging of siblings into sub-trees need to occur in lock step to ensure isomor-

Algorithm 2 MERGEOVERLAP($C.i, j$)

Input: A set C of common leaf labels between application-specific hierarchy i and j.

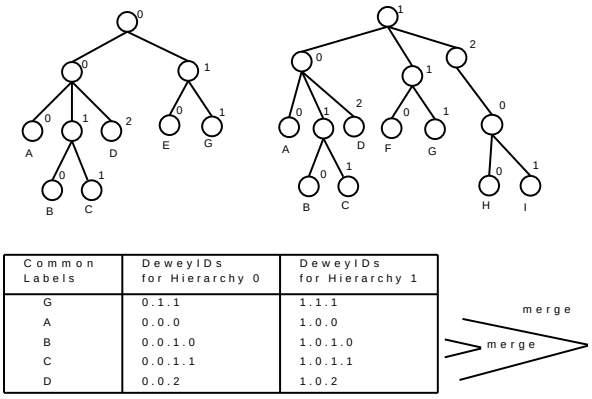
Output: A set of node ID pairs $\{(u_i, v_j)\}$ of the root nodes of the maximal overlapping sub-hierarchies in i and j.

1: Construct $M \leftarrow \{(u_i, v_j) : l_i(u_i) = l_j(v_j) = c, \forall c \in C\}$.
2: Sort M according to DeweyID(u_i).
3: **while** merge exists **do**
4: Scan through M, examine DeweyID(u_i) and DeweyID(v_j).
5: Let $M' \subseteq M$ be a set of node pairs that are siblings in both hierarchy i and j.
6: **if** M' covers all the sibling nodes in both hierarchy i and j **then**
7: Merge M' and replace M' with the pair of parent nodes in M.
8: return M

phism (Condition 1 of Definition 4). Line 6 checks for Condition 2 of Definition 4. Condition 3 of Definition 4 is automatically satisfied by construction of M. Groups of sibling pairs that can be merged are then removed from M and the pair of parent nodes inserted to M (Line 7). The while-loop terminates once no more merging can be applied.

The resulting pairs of nodes from the MERGEOVERLAP procedure are then returned and stored in $\mathcal{M}(i, j)$. As mentioned in the overview (Section 2.1), the equivalence or overlap information contained in $\mathcal{M}(i, j)$ are stored in a relational catalog table. The off-line phase need only be run whenever the hierarchies change. In most industrial application, hierarchies are seldom modified; however, new application-specific hierarchies may be added. If only new hierarchies are added, the algorithms in the off-line phase can be modified slightly so that only comparisons between the new hierarchies and the old hierarchies are performed, since the overlaps associated with the old hierarchies have not changed.

Complexity. The complexity of Algorithm 2 is $O(w \times d)$ time, where w is the maximum number of leaf nodes any application-specific hierarchy can have, and d is the maximum depth that any application-specific hierarchy can have. The while-loop in Line 3-7 executes for at most d iterations and each iteration takes $|C| = |M| = O(w)$ time. The complexity of Algorithm 3 is $O(|H| + |p|log|p| + |H_I|^2 \times w \times d)$, where H_I is the set of hierarchies that overlap with some other hierarchy. The $|H|$ term is due to the inverted index construction in Line 2 and the $|p|log|p|$ term is due to



Common Labels	DeweyIDs for Hierarchy 0	DeweyIDs for Hierarchy 1
G	0.1.1	1.1.1
A	0.0.0	1.0.0
B	0.0.1.0	1.0.1.0
C	0.0.1.1	1.0.1.1
D	0.0.2	1.0.2

Figure 4: An example of how the procedure MERGEOVERLAP works.

the sorting of the $|p|$ inverted lists by size in Line 3. The complexity of the nested for-loop in Line 5-6 is bounded by the number of comparisons between all pairs of hierarchies from H_I. If there are no overlaps among the hierarchies at all, H_I will be the empty set.

2.3. On-line Phase: Rewriting Queries

The on-line phase is invoked whenever an aggregation query that is based on an application-specific hierarchy is processed. Since the on-line phase attempts to rewrite the aggregation query using pre-computed results from other hierarchies, it requires that the off-line phase has been successfully executed and the equivalence information stored in the `Overlap` catalog table.

OLAP queries in data marts are specified as summations over the leaf nodes of an application-specific hierarchy. For example, a query on the application-specific project hierarchy of Figure 2(c) may be to sum all the expenses associated with the following leaf nodes of that hierarchy,

$$\begin{aligned} query \quad = \quad & AccountingTeam2 + SalesTeam \\ & + HRDept + AccountingTeam1 \\ & + MarketingTeam + ITEmployee9. \end{aligned}$$

Note that because the queries are developed based on the application-specific hierarchy, the summations typically follow the structure of the hierarchy.

Rewriting query using views is a well-studied problem [13, 14]; however, the problem that we address here is unique in that aggregation follow certain hierarchies and the equivalences may be at any level of a hierarchy.

Algorithm 3 sketches our algorithm for finding

Algorithm 3 REWRITEQUERY(Q, h, $\mathcal{M}$)

Input: A query Q on application-specific hierarchy h, and the set of overlaps $\mathcal{M}$

Output: A set of alternate query formulation

```
1:  QN ←MERGEQUERYNODES(Q, h)
2:  H ←find all hierarchies in M that overlap with h
3:  H ←eliminate hierarchies that do not have precomputed
    results
4:  /* check all alternate hierarchies for possible rewrites */
5:  for all i ∈ H do
6:     for all (u_h, v_i) ∈ M(h, i) do
7:        for all q ∈ QN do
8:           if q is covered by u_h then
9:              v_q ←find matching node for q in v_i.
10:             Alternate(q) ←Alternate(q) ∪ v_q
```

rewrites to an aggregation query. The input to our rewriting algorithm consists of

- a query Q, a set of leaf labels that identify rows in a fact table to be aggregated,
- the application-specific hierarchy h on which the query is based, and
- the equivalence information $\mathcal{M}$ in the `Overlap` catalog table that is discovered by the off-line phase.

The first step in the rewriting algorithm invokes the procedure MERGEQUERYNODES outlined in Algorithm 4 to convert the query (a set of leaf labels) into a set of sub-tree root nodes in the hierarchy h of which leaf nodes cover the query. The rewriting algorithm then finds H, the set of hierarchies that overlap with h, using the `Overlap` table (Line 2). Since not every hierarchy that overlaps with h have precomputed results in cache or materialized views in the database, such hierarchies can be pruned from H (Line 3). Next, we examine the equivalence node pairs for each candidate hierarchy in H (Line 5-6), and check if the equivalence nodes cover any of the query nodes (Line 7-8). This subsumption check can be done very efficiently using node labeling techniques such as interval labeling or Dewey labeling. Since node IDs in our system are Dewey IDs, the subsumption check reduces to checking for prefix match of the Dewey IDs. If the node u_h in the equivalence pair that is associated with the hierarchy h does indeed subsume the query node, we then the node in the equivalent sub-tree of v_i that is equivalent to the query node q (Line 9) and add that node to the alternate list for query node q (Line 10).

We now describe Algorithm 4 for transforming the query, a set of leaf labels, to a set of covering sub-tree

Algorithm 4 MERGEQUERYNODES(Q, h)

Input: A query Q, and the associated application-specific hierarchy h

Output: The set of hierarchy nodes QN that represent the query.

```
1:  /* construct list of hierarchy nodes associated with the
    query */
2:  N ← ∅
3:  for all term t in the query Q do
4:     Let I(t) be the set of nodes from h with label t
5:     N ← N ∪ I(t)
6:  /* merge nodes in N into sub-trees according to h*/
7:  Sort N by the DeweyID of each node
8:  while merge exists do
9:     Scan through N, examine DeweyIDs of each node.
10:    Let N' ⊆ N be a set of nodes that are siblings in hier-
       arhcy h.
11:    if N' covers all the sibling nodes in hierarchy h then
12:       Merge N' and replace N' with the parent node in
          M.
13: /* find sub-trees in N that covers the query */
14: QN ← ∅
15: while Q is not empty do
16:    Pick the sub-tree n ∈ N according to some greedy set
       cover heuristics
17:    QN ← QN ∪ n
18:    Q ← Q − LeafLabels(n)
19: if there are overlaps, subtract overlapping portion to avoid
    duplicate aggregation.
20: return QN
```

root nodes. The first step converts each leaf labels into its associated leaf node ID in the hierarchy h (Line 2-5). Note that each leaf label may map to more than one leaf node. The set of leaf node IDs N is then sorted according to the Dewey ID (Line 7). Recall that sorting Dewey IDs will group sibling nodes together. The while-loop in Line 8 iterates through the sorted node IDs and attempts to group sibling nodes into subtrees according to their Dewey ID. A group of sibling nodes are merged and replace with their parent node ID only if the parent has no other siblings. The while-loop terminates when no more merging is possible. At this point, the set N contains node IDs of the root nodes of sub-trees that cover the query. In fact, we allow more than one sub-tree to cover the same portion of the query. To find a minimum subtree cover, we employ a standard greedy, heuristic-based set cover algorithm (Line 15-18), since the problem is known to be NP-hard.

Complexity. The running time complexity of Algorithm 4 is $O(|Q| \times depth(h))$, because each iteration of the while-loop in Line 8-12 takes at most $O(|Q|)$

time and there are at most $O(depth(h))$ iteration (a leaf node can at most be merged up to the root of the tree). The complexity of the three nested for-loops in Algorithm 3 is $O(|H| \times \max_{i \in H} |i| \times |Q|)$, where $|i|$ denotes the size or number of nodes in hierarchy i. Hence, the overall complexity of the on-line phase is $O(|Q| \times depth(h) + |H| \times \max_{i \in H} |i| \times |Q|)$

3. Experiments

We prototyped our solution using Perl and investigated the effectiveness and the scalability of the proposed algorithms. This section presents highlights from our experimental study.

Data Generation. In order to understand the scalability of our algorithms, we generated synthetic data with different sizes and characteristics. A data set consists of a set of hierarchies or trees whose leaf labels are taken from a controlled vocabulary. Moreover, we need to synthesize controlled overlaps among the trees. Our approach is to first generate a collection of 100 random trees to be used as shared sub-trees. We then generate the trees for the data set such that it will include a subtree from the collection of shared sub-trees with a configurable probability $sharedprob$.

A random tree is generated as follows. Starting from a node, we generate a random probability and choose to expand the current node with a configurable probability $expandprob$. If the current node is a leaf, it is assigned a leaf label. Otherwise, a random number bounded by $maxfanout$ is generated for the number of children. The procedure is then called recursively for each child of the current node. The recursion terminates when $maxdepth$ has been reached. For generating random trees that share some common sub-trees, we add an addition step when expanding a node. A random probability is generated so that with probability $sharedprob$ a shared sub-tree should be used for expansion, and with probability $1 - sharedprob$ the node is expanded normally. If the current node is to be expanded with a shared sub-tree, a random shared sub-tree is picked from the collection without replacement.

We generated several of such data sets and verified that our off-line phase algorithm discovered all the shared sub-trees that were injected into the data set. We now describe some of the scalability results in detail.

Varying the number of hierarchies. In this experiment we study how the off-line phase algorithm scales with the number of trees in the data set. The $maxfanout$ is set at 5, the $maxdepth$ at 16, the $expandprob$ and $sharedprob$ both at 0.8. For each setting for the number of hierarchies, we generate 10 random data sets and ran the off-line phase on these data sets. The average running time over the 10 runs is then measured. Note that the running time includes reading the hierarchies from disk. Figure 5(a) shows our measurements. Although the running time of the off-line phase appears to be super-linear in terms of the number of hierarchies, if we replot the measurements using the number of pairs of shared sub-hierarchies as the X-axis, we observe in Figure 5(b) a linear relationship with the number of shared pairs which is also the size of the output.

Varying the hierarchy fanout. The number of hierarchies is set at 200, the $maxdepth$ at 16, the $expandprob$ and $sharedprob$ both at 0.8. Figure 5(c) shows the running time of the off-line phase versus the hierarchy fanout characteristics. Our results show that the running time of the off-line phase is not sensitive to the fanout characteristics of the hierarchies. This is a nice property for an algorithm operating on hierarchies.

Varying the hierarchy depth. The number of hierarchies is set at 200, the $maxfanout$ at 10, the $expandprob$ and $sharedprob$ both at 0.8. Figure 5(d) shows the running time of the off-line phase versus the hierarchy depth characteristics. Our results show that the running time of the off-line phase is not sensitive to the depth of the hierarchies. This is a nice property for an algorithm operating on hierarchies.

Varying the probability of expansion. Recall that the probability of expansion controls how dense each random tree is going to be. In this experiment the number of hierarchies is set at 200, the $maxfanout$ at 10, the $maxdepth$ at 16, and $sharedprob$ at 0.8. Figure 5(e) shows the running time of the off-line phase versus the hierarchy expansion probability. Our results show that the running time of the off-line phase is not sensitive to the depth of the hierarchies.

Varying the probability of sharing. Recall that the probability of sharing controls how much overlap there will be among the hierarchies in the data set. In this experiment the number of hierarchies is set at 200, the $maxfanout$ at 10, the $maxdepth$ at 16, and $expandprob$ at 0.8. Figure 5(f) shows the running time of the off-line phase versus the hierarchy sharing probability. Somewhat surprisingly, our results show that the running time of the off-line phase is not sensitive to the sharing probability of the hierarchies.

4. Related Work

In traditional OLAP environments [2], dimension hierarchies are regular balanced trees that are typically

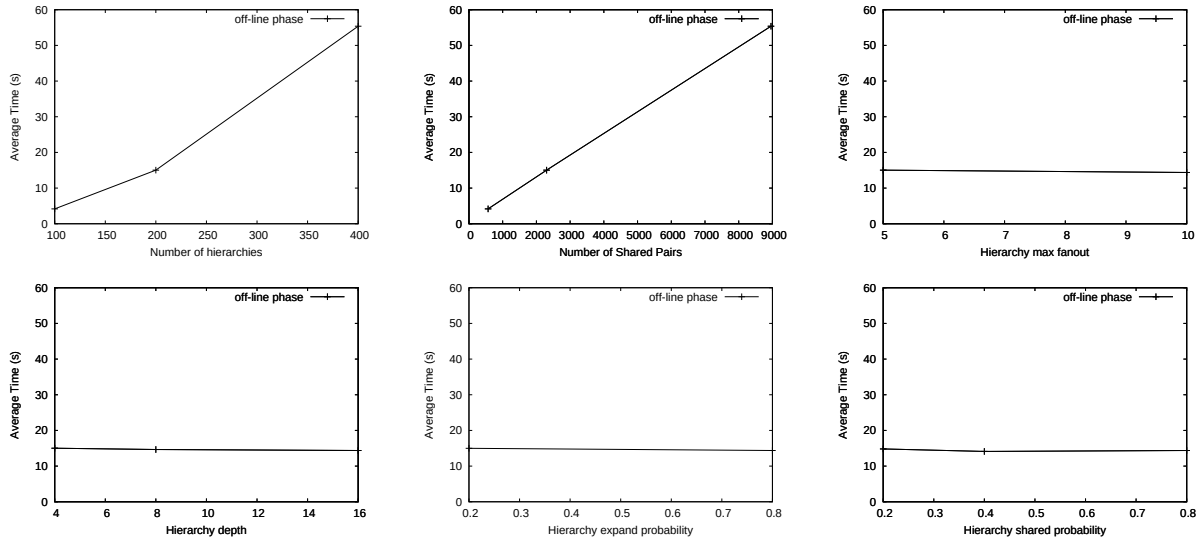


Figure 5: Running time of off-line phase against different data set characteristics.

stored as flattened dimension tables. A dimension table for a dimension hierarchy consists of one column per level of the hierarchy. The values in each column represent the distinct nodes at the level associated with that column. Roll-up aggregations on a dimension hierarchy can then be computed as a group-by on a join between the dimension table and the fact table. Such aggregation queries are typically optimized by precomputing the aggregations and storing them in a summary table or materialized view. Much research has been done on the problem of view selection [7, 19, 8], the problem of rewriting a query to exploit a set of materialized views [3, 4, 13, 14], and the problem of view maintenance [1, 22, 6]. These previous work address flat relational views, regular and balanced hierarchies, and cube views. Our work is complementary in that we address specifically sub-tree overlaps among irregular and unbalanced dimension hierarchies. Once the overlapping sub-trees are identified, the techniques from these previous work can be used to create views on the sub-trees.

The Treescape [16, 15, 16] system proposed by Pedersen et al. does handle irregular dimension hierarchies. The focus of their work is on algorithms for determining what to pre-aggregate. The key idea in Treescape is to transform hierarchies so that they have certain nice properties that allow better pre-aggregation to be done. Our work is orthogonal to Treescape: we do not address what to pre-aggregate, but what is shared between hierarchies and how we can exploit the shared sub-trees if pre-aggregations on them exists.

Xu et al. [20] proposed a method for computing closest common subexpressions for view selection problems that works at the level of relational algebra and hence SQL statements. Our work differs in that we identify common sub-hierarchies among hierarchies.

Apart from academic research, the Master Data Management (MDM) [12] approach can be used as an alternative solution to the problem reported in this paper. MDM is a set of processes and tools for maintaining "a single version of the truth" in an enterprise and entails significant organizational change. In principle, MDM and its associated software tools [9, 17, 10] can be used to maintain a single set of application-specific hierarchies with no overlaps. On the other hand, MDM is not very useful if the problem already exists.

The problem addressed by the off-line phase of our approach resembles the problem of finding common sub-trees addressed by [21]; however, our definition of "common sub-tree" is quite different Zaki's and his techniques do not extend easily to our problem.

The on-line phase of our approach is related to the problem of query rewriting using views [13, 14]. The key difference is that in the traditional setting, views equivalences are flat relations, whereas in our setting the partial, sub-tree equivalences identified in the off-line phase needs to be incorporated when searching for a candidate view to rewrite the query.

Previous research on partial sums on OLAP data cubes is also related to our work in that each matching sub-hierarchy can be viewed as a partial sum. The

difference is that previous work do not exploit any tree structure on the full and partial aggregations. Our work, on the other hand, fully exploits the tree structure in determining the partial sums common to multiple aggregations.

5. Conclusion

In this paper we describe techniques to optimize OLAP aggregation queries defined on an application-specific hierarchy by leveraging precomputed aggregations on other hierarchies. This is done by detecting common sub-structures in the different hierarchies and by providing a rewriting algorithm to exploit any precomputations on these shared structures. Our techniques result in significant savings of computation and faster query response times. While current technology like MDM is very good at keeping track of the many hierarchies that could be defined on a dimension and resolving them upfront, it does not do a good job of finding common substructures when the hierarchies have already been defined or in determining a more efficient way to express the derived accounts in terms of the summary accounts. It is in situations like this that our techniques help in alleviating the performance bottlenecks.

6. References

[1] J. A. Blakeley, N. Coburn, and P.-V. Larson. Updating derived relations: detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[3] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200. IEEE Computer Society, 1995.

[4] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In *EDBT '96: Proceedings of the 5th International Conference on Extending Database Technology*, pages 167–182. Springer-Verlag, 1996.

[5] L. Gong, M. Olivas, C. Posluszny, D. Venditti, and G. McMillan. Deliver an effective and flexible data warehouse solution, Part 2: Develop a warehouse data model. *IBM Developerworks*, July 2005. `http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0507gong/`.

[6] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[7] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT '99: Proceedings of the 7th International Conference on Database Theory*, pages 453–470, 1999.

[8] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *TKDE*, 17:24–43, 2005.

[9] `http://dev.hyperion.com/products/mdm/`.

[10] `http://www-306.ibm.com/software/data/masterdata/master-data-management.html`.

[11] L. Lim and B. Bhattacharjee. Optimizing hierarchical access in OLAP environment. In *ICDE*, pages 1531–1533, 2008.

[12] D. Loshin. *Master Data Management*. Morgan Kaufmann Publishers Inc., 2008.

[13] C.-S. Park, M.-H. Kim, and Y.-J. Lee. Rewriting OLAP queries using materialized views and dimension hierarchies in data warehouses. In *ICDE*, pages 515–523, 2001.

[14] C.-S. Park, M.-H. Kim, and Y.-J. Lee. Finding an efficient rewriting of OLAP queries using materialized views in data warehouses. *Decision Support Systems*, 32(4):379–399, 2002.

[15] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson. Extending practical pre-aggregation in on-line analytical processing. In *VLDB*, pages 663–674, 1999.

[16] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson. The TreeScape system: Reuse of pre-computed aggregates over irregular OLAP hierarchies. In *VLDB*, pages 595–598, 2000.

[17] `http://www.sap.com/platform/netweaver/components/mdm/index.epx`.

[18] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

[19] S. R. Valluri, S. Vadapalli, and K. Karlapalem. View relevance driven materialized view selection in data warehousing environment. In *ADC '02: Proceedings of the 13th Australasian database conference*, pages 187–196. Australian Computer Society, Inc., 2002.

[20] W. Xu, D. Theodoratos, and C. Zuzarte. Computing closest common subexpressions for view selection problems. In *DOLAP '06: Proceedings of the 9th ACM international workshop on Data warehousing and OLAP*, pages 75–82. ACM Press, 2006.

[21] M. J. Zaki. Efficiently mining frequent trees in a forest. In *KDD 2002*, pages 71–80. ACM Press, 2002.

[22] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327. ACM, 1995.