Efficient Roll-Up and Drill-Down Analysis in Relational Databases

Min Wang
Duke University
minw@cs.duke.edu

Bala Iyer
IBM Santa Teresa Lab
balaiyer@vnet.ibm.com

Abstract

Large collections of data, e.g. archives of POS (point-of-sale) in the consumer packaged goods as well as in prescription drugs, are typically iteratively classified by a hierarchy [6] on the members of some attributes. Most of the fortune 500 businesses have found ways to consolidate the data archive into a relational database manager. One of the most common steps in "investigative data exploration" is the navigation of the hierarchy to glean valuable insights about the state of the business. The exploration answers "what if" questions, e.g., what-if the retailing of toys were discontinued in all stores and an investment made for building a retail organization for personal computers. To answer such questions, we are required to take the data associated with the lowest level of the hierarchy (e.g., sales) of individual items and sum it up to the ancestor at the highest level of the hierarchy, an operation referred to as **roll-up**. A human analyst, with the aid of tools, usually narrows down to one or more ancestors, then does analysis on the children. The later operation often involves obtaining sums for the children which is referred to as **drill-down**. Analysis would proceed recursively. The problem is to support efficient roll-up and drill-down on the data collection stored in the relational database management system. Two solutions are considered. In one, the hierarchy is stored as multiple map tables. In the second, members of the hierarchy are labeled in a manner that is sensitive to the hierarchy. Results of performance experiments comparing the two are given. Method using labeling has a factor of 3 advantage over the other. We also give the reasons why this may be so.

1 Introduction

Aggregate queries are used frequently in decision support applications, where the basic goal is to collect various information from *detail (base)* tables [3]. One important characteristics of many such applications is that the aggregation is actually done on a tree hierarchy [6]. For such hierarchy structure, decision making is usually done in two phases. In the first roll-up phase, information is gathered from lower levels into upper levels. In the second drill-down phase, gathered information is used to guide branching from upper levels to lower levels so that particular sets of base table items are further studied.

The traditional way to do roll-up and drill-down analysis is by storing the hierarchy structure as multiple map tables and computing the aggregations through joins using those maps and the detail tables. Since detail tables are so large in data warehousing environments, the joins are very expensive.

In this paper, we propose another method to do roll-up and drill-down analysis. Instead of working with joins involving large tables, we process the hierarchy structure directly. We label the members of the hierarchy tree in such a way that all necessary tree information is known through the label. Using our method, one does not need to do any join. We refer to this alternative method as **labeling method**. Experiments show that the performance of the labeling method is much better than that of the traditional join method.

The paper is organized as follows. In Section 2, we state the problem to be solved. In Section 3, we discuss the join method for doing "roll-up" and "drill-down" analysis. In Section 4, we outline our labeling method. Section 5 contains our experiment results and we give our conclusion in Section 6.

2 The Problem

Consider a data warehouse with sales data for a large retail store. Conceptually, the store has a hierarchy structure as in Figure 1.

The detail (base) table contains the amount of sales by featured products for sales over a long period of time. The detail table has the following schema:

```
sales(item_id, amount)
```

Two interesting and important data mining techniques are *roll-up* and *drill-down* analysis. On a hierarchy structure, information at the lower levels is collected

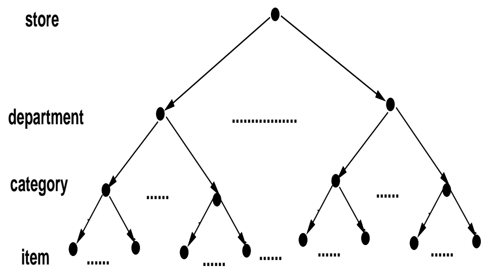

Figure 1: Hierarchy for the retail store data

bottom-up into upper-levels, and the collected information is then used to select particular branches that have specified features, this time in a top-down fashion.

For example, for the retail store data mentioned above, we may want to find out the total amount of items sold by each department, and then single out the "most important" department, e.g., the department with the largest sale amount. Furthermore, we may wish to obtain the sales amount for each category in that department. If there are more levels between categories and items, we may go further and choose the most important category and drill down one more level. We can repeat this process until we reach the level just above the leaves. The procedure is depicted in Figure 2, by following the arrows from top to bottom. We are interested in finding out the sales amount for all the highlighted nodes in a 4-level hierarchy.

The following implementation alternatives are possible for such problem:

(1) full pre-computation, where aggregates are pre-computed and stored in every tree node.
(2) partial pre-computation, where aggregates are pre-computed and stored in selected tree nodes, and
(3) no pre-computation, where no aggregates are pre-computed and stored.

During data explorations, it is possible during roll-

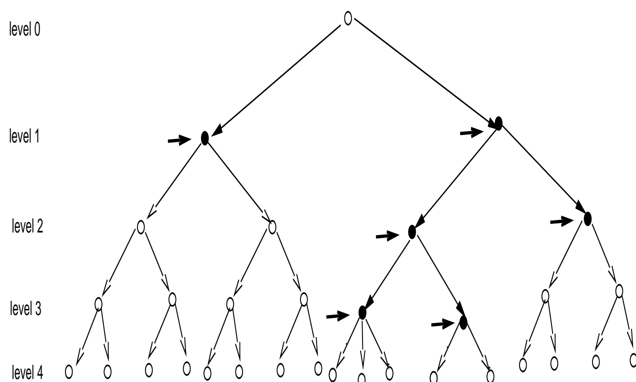

Figure 2: Roll-up and drill-down analysis

up and drill-down, other analysis may occur on an unpredictable subset of data (e.g., customers with income more than $100,000 and age older than 45 and ZIPCODE is 12345). This occurs when a segmentation algorithm is used in conjunction with roll-up and drill-down. Thus, it is desirable to keep as much detail as possible without any pre-computation. The alternatives are (1) and (2) complemented with efficient materialized view maintenance mechanism [7]. Handling large volumes of updates is still a challenge for approach (1) and (2). The last approach is viable on existing commercial DBMSs today, so we limit our discussion to the third approach.

3 Join Method

A natural way to do roll-up and drill-down analysis is through joins among detail and map tables.

Let's consider the department store example. Suppose we have the following relations in the database:

```
sales(item_id, amount)
map_c2i(cate_id, item_id), unique in item_id
map_d2c(dept_id, cate_id), unique in cate_id
```

where `sales` is the detail table, `map_c2i` and `map_d2c` are two map tables.

We can do the analysis using the following SQL queries. First, we define a view that maps `item` to `category` using a join:

```
CREATE VIEW cate_sales(cate_id, amount) AS
SELECT cate_id, sum(amount)
FROM map_c2i, sales
WHERE map_c2i.item_id=sales.item_id
GROUPBY cate_id;
```

Then we define another view using another join to map `category` to `department`:

```
CREATE VIEW dept_sales(dept_id, amount) AS
SELECT dept_id, sum(amount)
FROM map_d2c, cate_sales
WHERE map_d2c.cate_id=cate_sales.cate_id
GROUPBY dept_id;
```

Instantiation of `dept_sales` will give us the total amount of items sold in each department and completing roll-up. At this moment, we can obtain the sales amount for each department by issuing a simple SELECT query on this view.

Next we proceed with the drill-down phase. First we choose our "importance" measurement, e.g., we consider the department with the most sales amount as the most important one. Let its `dept_id` be x.

To obtain the sales amount for each category in department x, we need to do one more join:

```
CREATE VIEW x_sales(cate_id, amount) AS
SELECT cate_id, amount
FROM map_d2c, cate_sale
WHERE map_d2c.cate_id=cate_sale.cate_id
      and map_d2c.dept_id=x;
```

If there are more levels, we need to do more joins to complete the drill-down analysis. It's easy to see that the cost of the whole process is dominated by the join involving the relation `sales`, since it is done between the huge detail table and the largest map table which contains the same number of rows as the unique items in detail.

4 Labeling Method

Instead of storing the mappings between each adjacent levels as map tables and performing the expensive join operations to do the analysis, it is possible to process the hierarchy structure directly. We label the nodes in the hierarchy tree in such a way that the parent-child relationship can be obtained easily from the label.

The benefits from encodings for roll-up and drill-down queries of OLAP have been discussed by Wong et. al. in [9]. Various encoding method and benefits are discussed. In [5], more encoding options are pointed out and the use of gray codes are suggested if there is a natural ordering on the members. In our problem, the hierarchy imposed a partial order on the members and we need to choose an encoding method that can capture this partial order, e.g., the parent-child relationship.

Particularly, we label each node in the hierarchy tree according to a post-order traversal [1]. Figure 3 gives the labeling for a simple hierarchy tree that contains only 17 nodes.

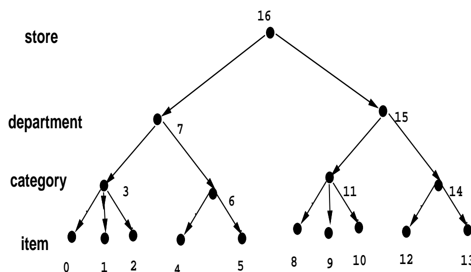

Figure 3: Post-order labeling of a small hierarchy tree

A post-order labeling has the following property: for any node with label j, if the smallest label of its descendents is i, then $i < j$ and it has exactly $j-i$ descendents with labels from i to $j-1$. That is, the range $[i : j-1]$ gives the labels of all its descendents. For simplicity, we will call a node by its label.

For example, all the categories and items belonging to the second department (node 15) are labeled as a number in the range [8:14].

Thus, any node j is associated with a range r_j which gives the labels of all it's descendents.

In the detail table, instead of storing `item_id` directly, we store the label for each item. So the schema for the detail table is:

`sales(item_label, amount).`

After we label the hierarchy tree, we can use label ranges to collect information and speed up the roll-up and drill-down analysis. We make use of an object extensions to DB2, the **user-defined function (udf)** [8, 2] to do that.

User-defined function is a new DB2 feature provided by DB2 version 2 [2, 10]. In DB2 V2 system, the functions available for use in SQL statements extend from the system built-in functions such as `avg`, `min`, `max`, `sum` to more general categories, such as user-defined functions (udfs). Particularly, an external udf is a function that is written by a user in a host programming language. The `CREATE FUNCTION` statement for an external function tells the system where to find the code that implements the function. For detail information on udf, see [2, 10].

Let's define a udf `f(T, level_num, node_num, item_label)`, where `T` is a hierarchy tree, `level_num` an integer indicating a level number, `node_num` an integer indicating the position of a node at a given level, and `item_label` an integer representing an item. The output of the function is the range (actually a numerical range number) which contains the numerical label for that item. Since `T` is clear from the context, we will omit it.

Once we have such a function, the roll-up and drill-down analysis can be done as follows. We will use the following query pattern (denoted by `q(level_num, node_num, item_label)`):

```
SELECT f(level_num, node_num, item_label),
       sum(amount)
FROM sales
GROUPBY f(level_num, node_num, item_label)
```

We process the hierarchy tree level by level in a top-down fashion. For level 0, we have only one root node. We scan the detail table once by issuing the query `q(0, 1, item_label)`. After the scan is done, we obtain the sum of sales amount for each department.

Now we choose the most important department, say the `k`th department, and go to the next level, level 1.

For level 1, we scan the base table again by call the query `q` with `level_num=1` and `node_num=k`. After the scan is done, we obtain the sales amount for each category in the `k`th department. This completes the analysis.

In the implementation of the user-defined function (udf) `f`, at the first call, go through initialization. We read the proper portion of the labeled hierarchy tree (`level_num` and `node_num` will tell us what is the proper portion). Then from the labeled tree and `level_num` and `node_num`, we could easily find the corresponding label value ranges for the subtree rooted at each child of the interested node. We store such ranges in the scratch-pad of the udf. In all the calls to the udf, we just search the ranges in the scratch-pad to find the one that contain the current `item_label`. If the current record falls into the `i`th range, then `i` is the return value of the udf. If the current record doesn't fall into any of the range, null is the return value. The following pseudo-code gives an overview of the udf `f`:

```
f(level_num, node_num, item_label)
  if (calltype is FirstCall) {
    read in the proper portion of the
      encoded hierarchy tree according
      level_num, node_num;
    get the code range for each subtree
      that rooted at any child of the
      interested node;
    denote those ranges by r_1, r_2, ..., r_n;
    store r_1, r_2, ..., r_n in the scratch-pad of f;
  }
  if (calltype is FirstCall or MiddleCall){
    output=null;
    if (∃ r_i s.t.  item_label∈ r_i)
      output=i;
    return;
  }
  if (calltype is LastCall)
    free the memory allocate for scratch-pad;
```

Note those ranges are in sorted order at any level, so we could use binary search to find the corresponding range for a record when the out degree of the interesting node is large. This guarantees that the execution time of the udf for each record will increase only logarithmically with respect to the out-degree of the hierarchy.

5 Experiments

We compare the performance of the labeling method with that of the join method on a hierarchy tree of 6 levels. The hierarchy we use is a balanced tree with an out-degree of 10 at each non-leaf node. In total we'll have 1 million records in our detail table. In a CPG (Consumer Packaged Goods) application, this corresponding to the retailer tracking 1 million unique SKUs (Stockable Unit). This is the situation at large US retailers today, because of highly differentiated products (e.g., dark, medium or light black, temporary or permanent hair dye from various manufactures in various packagings and sizes) and accelerated product cycles.

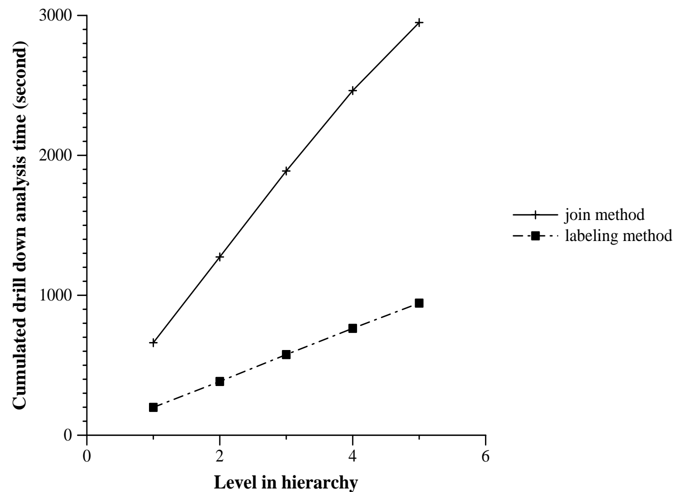

Figure 4: Performance comparison for the two methods

For the join method, We generate the base table `sales` based on a large file. The file contains 1 million rows. Each row has two numbers. The first column in the file, which is the `item_id`, is a random permutation of the numbers from 1 to 1,000,000. The second column (`amount`) is 1 million random numbers which are smaller than some predefined max account. We load such file into the table `sales`.

Then we generate the map table for the mapping from level 5 to level 6 (leaf level). The first column is 10 random permutations of the number from 1 to 100,000, and the second column is a random permutation of the number from 1 to 1,000,000.

Similarly, we can generate the other map table for each adjacent level pair. We have 5 such map tables, each contain 2 columns. The table for the mapping from level i to level $i+1$ has a total of 10^{i+1} rows, where $i = 1, 2, 3, 4, 5$.

Using the base table and the 5 map tables, we first use join method as we described in Section 3 to do the roll-up and drill-down analysis and measure the time for each step.

For the labeling method, we only need to generate the base table according to the encoded hierarchy.

Our experiments are conducted on an IBM RS/6000 workstation running AIX level 4.1.3. and DB2 version 2.1.1. The performance of the two methods is shown in Figure 4. We plot the cumulative response time to do the roll-up and drill-down analysis on the experimental data. A point (l, t) on the graph indicates that it takes t seconds to finish the analysis for levels $1, 2, \ldots, l$. We could see easily from the figure that the labeling method has a factor of 3 advantage over the join method.

Our experiments suggest that response time is domi-

nated by I/O on the detail table. Hence for both solutions considered, the response is linear in the number of levels of the hierarchy that are explored. Every level, in the absence of pre-computation, requires access to the detail table. We investigated the reason for the factor of 3 advantage for the method using labels. For the labeling method, exploration of every level resolves to one I/O pass over detail. In the absence of labeling or pre-computation, the join method requires the detail to be read, sorted into runs, runs merged for join processing. Although amortized over many rows, each row needs to be read from disk, written as part of a sorted run, then read again during merging and join. Since I/O bandwidth is saturated, this appears to be the reason for tripling of the response time.

The above analysis is consistent with the access plan for the join method queries. To confirm, we built a table mapping detail items to level 1 and then did a join between this table and the detail. The response execution time for doing the join was almost the same as roll-up or drill-down by one level (about 600 seconds in our experiments).

Current computer systems have I/O speed several factor higher than the I/O speed of the system we used. In addition, we have massive parallelism, like in IBM's

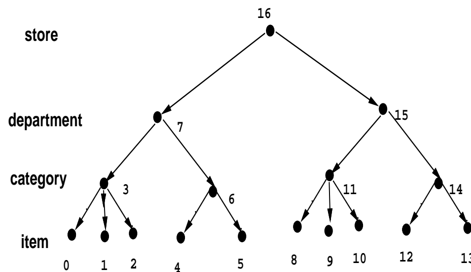

Figure 5: Post-order labeling of a small hierarchy tree

SP2 system that uses an MIMD architecture to interconnect 100's of processors. On such computer systems, it is expected that roll-up and drill-down would occur at the "speed of the thought".

6 Conclusion

By means of a simple and intuitive benchmark proposed in this paper, we illustrate the problem of "roll-up" and "drill-down" analysis on the detail members of a hierarchy. We use two different methods to perform the operations and demonstrate the relative efficiency of the post-order labeling method.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] Don Chamberlin. *Using the New DB2: IBM's Object-Relational Database System*, Morgan Kaufmann, 1996.

[3] A. Gupta, V. Harinarayan, and D. Quass. "Aggregate-Query Processing in Data Warehousing Environments," *Proc. of the 21st Int'l Conference on Very Large Database*, Zurich, Switzerland, 1995.

[4] V. Harinarayan. *Query Processing in Data-Warehousing Environments,* PhD thesis, Stanford University, 1996.

[5] V. Harinarayan et. al. "Encoded-Vector Indices for Decision Support and Warehousing," *IBM STL Internal Report*, Sept. 1995.

[6] R. A. Kevin, D. Kalyanaraman, and D, J. Howard. "Product Hierarchy and Brand Strategy Influences on the Order of Entry Effect for Consumer Packaged Goods," *Journal of Product Innovation Management*, Vol. 13, No. 1, Jan. 1996.

[7] I. S. Mumick and A. Gupta. *Proceeding of the Workshop on Materialized Views: Techniques and Applications*, Montreal, June 1996.

[8] M. Stonebraker and L. A. Rowe. "The Design of Postgres," *Proc. of SIGMOD 1986.*

[9] H. K. T. Wong et. al. "Bit Transposed Files,", *Proc. of VLDB 1985,* Stockholm, 1985.

[10] *IBM DATABASE 2 Application Programming Guide-for common servers*, version 2.