2009 IEEE International Advance Computing Conference (IACC 2009)
Patiala, India, 6–7 March 2009

Role of Materialized View Maintenance with PIVOT and UNPIVOT Operators

A.N.M. Bazlur Rashid, M. S. Islam

Institute of Information and Communication Technology
Bangladesh University of Engineering and Technology
Dhaka – 1000, Bangladesh
bazlur.rashid@yahoo.com, mdsaifulislam@iict.buet.ac.bd

***Abstract* – PIVOT and UNPIVOT are very useful operators for on-line analytical processing (OLAP) applications and querying sparse datasets. These operators exchange rows and columns which can be efficiently used for data presentation in data warehouse environment. Materialized view is another important issue for data warehouse and OALP that process user queries effectively. The process of updating materialized view in response to change is a great challenge in data warehousing environment. Maintenance of materialized views efficiently with PIVOT and UNPIVOT operators is also a challenge today in the field of RDBMS as well as data warehouse. In this paper, we discuss capabilities of PIVOT and UNPIVOT operators; materialized view maintenance, view maintenance work with PIVOT and UNPIVOT operators and finally focuses on the research trends in view maintenance.**

I. INTRODUCTION

A data warehouse is a subject-oriented, integrated, nonvolatile collection of data that is used primarily in organization decision making. It is typically multidimensional. For example, in a sales data warehouse, the product, location and time of sales might be some of the dimensions of interest. These dimensions are often hierarchical, for example, the location dimension may be organized as a city-state-country hierarchy. Many complex on-line analytical processing (OLAP) operations are designed to perform online analysis on such multidimensional data, such as drill down or roll up (decrease or increase the level of aggregation) in one or more dimension hierarchies, slice and dice (select one particular dimension value) and pivot (re-orient the multidimensional view of data) [1].

Pivot transforms a series of rows into a series of fewer rows with additional columns. Data in one source column is used to determine the new column for a row and another source column is used as the data for that new column. Unpivot provides the inverse operation, removing a number of columns and creating additional rows that capture the column names and values from the pivoted form. The pivoted form can be considered as a matrix of column of values while the unpivoted form is a natural encoding of a sparse matrix [2]. Fig. 1 demonstrates how PIVOT and UNPIVOT operators can transform data between pivoted and unpivoted table.

Pivoted Table

Month	Computer	Laptop	Scanner
Apr	500	550	700
May	525	600	800
Jun	NULL	650	900

Unpivot ↓ ↑ **Pivot**

Unpivoted Table

Item	Month	Quantity
Computer	Apr	500
Computer	May	525
Laptop	Apr	550
Laptop	May	600
Laptop	Jun	650
Scanner	Apr	700
Scanner	May	800
Scanner	Jun	900

Fig. 1: Action of PIVOT and UNPIVOT operation

Materialized views are the derived relations, which are stored as relations in the database. Materialization of views is one of the classical problems in data warehousing. Materialized views can be used for reducing query response time. Because of the query intensive nature of data warehousing, materialized views approach is quite promising in efficiently processing the queries. When a base relation is updated, all its dependent materialized views have to be updated in order to maintain the consistency and integrity of the database. The process of updating a materialized view in response to the changes in the base relation is known as view maintenance that incurs a view maintenance cost [3].

Most of the work on materialized view maintenance focuses maintenance of views with distributive functions such as SUM, COUNT and aggregation functions such as variance and regression. Many decision support queries also include various extracted, transformed and loaded (ETL), and OLAP operations, such as, PIVOT and UNPIVOT which subsequently involve functions such as SUM, COUNT. So the maintenance of materialized views with PIVOT and UNPIVOT operators is very important and also for the performance issues. Oracle database 11g introduces these two new operators, whereas Microsoft SQL Server already has these operators. There are very few works on the materialized view maintenance with PIVOT and UNPIVOT operators. In this

work we try to focus on the unique characteristics of these two operators and view maintenance with these operators in data warehouse environment.

II. PIVOT and UNPIVOT Operators

The PIVOT operator makes it easy to create aggregated cross-tabular output that condenses many rows into a compact result set useful for reports. For instance, input data holding sales of one month in each row can be pivoted into output holding twelve months in each row, with each month in its own column. By combining multiple input rows into each output row, PIVOT also enables inter-row comparison without a table self-join. The UNPIVOT operator reshapes data into a format useful for further relational operations. For example, if a source data set presents twelve months of sales values in each row, UNPIVOT can reshape each source row into twelve output rows, each holding one month of sales data. The unpivoted results are in a more normalized relational form than the source data, and they can be manipulated with simpler and more efficient SQL [4]. We describe the capabilities of PIVOT and UNPIVOT operators in the subsequent subsections in detail.

A. Pivoting Operations

The data returned by business intelligence queries is often most usable if presented in a cross tabular format. The pivot_clause of the SELECT statement lets you write cross tabulation queries that rotate rows into columns, aggregating data in the process of the rotation. Pivoting is a key technique in data warehouses. In it, multiple rows of input are transformed into fewer and generally wider rows in the data warehouse. When pivoting, an aggregation operator is applied for each item in the pivot column value list. The pivot column cannot contain an arbitrary expression. If we need to pivot on an expression, then we should alias the expression in a view before the PIVOT operation. The basic syntax is as follows:

```
SELECT ....
FROM <table-expr>
PIVOT (aggregate-function(<column>)
FOR <pivot-column>
IN (<value1>, <value2> ,..., <valuen>)
) AS <alias>
WHERE ....
```

To illustrate the use of pivoting, create the following view as a basis for later examples:

```
CREATE VIEW sales_view AS
SELECT prod_name product, country_name
country, channel_id channel,
SUBSTR(calendar_quarter_desc, 6,2) quarter,
SUM(amount_sold) amount_sold,  SUM(quantity_sold)
quantity_sold
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id
AND sales.prod_id = products.prod_id
AND sales.cust_id = customers.cust_id
AND customers.country_id = countries.country_id
GROUP BY prod_name, country_name,channel_id,
SUBSTR(calendar_quarter_desc, 6, 2);
```

The following statement illustrates a typical pivot on the channel column:

```
SELECT * FROM
(SELECT product, channel, amount_sold
FROM sales_view) S PIVOT (SUM(amount_sold)
FOR CHANNEL IN (3 AS DIRECT_SALES,
4 AS INTERNET_SALES,
5 AS CATALOG_SALES, 9 AS TELESALES)
) ORDER BY product;

PRODUCT DIRECT_SALES INTERNET_SALES
CATALOG_SALES TELESALES
---------- ---------- ---------- ---------- -------
-- ...
Internal 6X CD-ROM 229512.97 26249.55
Internal 8X CD-ROM 286291.49 42809.44
Keyboard Wrist Rest 200959.84 38695.36 1522.73
...
```

Note that the output has created four new aliased columns, DIRECT_SALES, INTERNET_SALES, CATALOG_SALES, and TELESALES, one for each of the pivot values. The output is a sum. If no alias is provided, the column heading will be the values of the IN-list.

B. Pivoting on Multiple Columns

We can pivot on more than one column. The following statement illustrates a typical multiple column pivot:

```
SELECT * FROM
(SELECT product, channel, quarter, quantity_sold
FROM sales_view)
PIVOT (SUM(quantity_sold)
FOR (channel, quarter)
IN ((5, '02') AS CATALOG_Q2,
(4, '01') AS INTERNET_Q1,
(4, '04') AS INTERNET_Q4,
(2, '02') AS PARTNERS_Q2,
(9, '03') AS TELE_Q3));

PRODUCT CATALOG_Q2 INTERNET_Q1
INTERNET_Q4 PARTNERS_Q2 TELE_Q3
------- ---------- ---------- ---------- ---------- -------
...
Bounce 347 632 954
...
Smash Up Boxing 129 280 560
...
Comic Book Heroes 47 155 275
...
```

Note that this example specifies a multi-column IN-list with column headings designed to match the IN-list members.

C. *Pivoting: Multiple Aggregates*

We can pivot with multiple aggregates, as shown in the following example:

```
SELECT *
FROM
(SELECT product, channel, amount_sold, quantity_sold
FROM sales_view)
PIVOT (SUM(amount_sold) AS sums,
SUM(quantity_sold) AS sumq
FOR channel IN (5, 4, 2, 9))
ORDER BY product;

PRODUCT 5_SUMS 5_SUMQ 4_SUMS 4_SUMQ
2_SUMS 2_SUMQ 9_SUMS 9_SUMQ
------------ ------ ------ ------ ------ ------ ------ ------ ------
O/S Doc Set English 142780.36 3081 381397.99 8044
6028.66 134
O/S Doc Set French 55503.58 1192 132000.77 2782
...
```

Note that the query creates column headings by concatenating the pivot values (or alias) with the alias of the aggregate function, plus an underscore.

D. *Distinguishing PIVOT Generated NULLs from NULLs in Source Data*

We can distinguish between null values that are generated from the use of PIVOT and those that exist in the source data. The following example illustrates nulls that PIVOT generates.

The following query returns rows with 5 columns, column prod_id, and pivot resulting columns Q1, Q1_COUNT_TOTAL, Q2, Q2_COUNT_TOTAL. For each unique value of prod_id, Q1_COUNT_TOTAL returns the total number of rows whose qtr value is Q1, that is, and Q2_COUNT_TOTAL returns the total number of rows whose qtr value is Q2.

Assume we have a table sales2 of the following structure:

```
PROD_ID QTR AMOUNT_SOLD
------- --- -----------
100 Q1 10
100 Q1 20
100 Q2 NULL
200 Q1 50

SELECT * FROM sales2
PIVOT (SUM(amount_sold), COUNT(*) AS count_total
FOR qtr IN ('Q1', 'Q2'));
PROD_ID "Q1" "Q1_COUNT_TOTAL" "Q2"
"Q2_COUNT_TOTAL"
------- ---- ---------------- --------- ----------------
100 20 2 NULL <1> 1
200 50 1 NULL <2> 0
```

From the result, we know that for prod_id 100, there are 2 sales rows for quarter Q1, and 1 sales row for quarter Q2; for prod_id 200, there is 1 sales row for quarter Q1, and no sales row for quarter Q2.

So, in Q2_COUNT_TOTAL, we can identify that NULL<1> comes from a row in the original table whose measure is of null value, while NULL<2> is due to no row being present in the original table for prod_id 200 in quarter Q2.

E. *Unpivoting Operations*

An UNPIVOT does not reverse a PIVOT operation. Instead, it rotates data from columns into rows. If we are working with pivoted data, an UNPIVOT operation cannot reverse any aggregations that have been made by PIVOT or any other means.

To illustrate unpivoting, first create a pivoted table that includes four columns, for quarters of the year:

```
CREATE TABLE pivotedTable AS
SELECT *
FROM (SELECT product, quarter, quantity_sold
FROM sales_view)
PIVOT (SUM(quantity_sold)
FOR quarter IN ('01' AS Q1,
'02' AS Q2, '03' AS Q3, '04' AS Q4));
```

The table's contents are the following:

```
SELECT *
FROM pivotedTable
ORDER BY product;

PRODUCT Q1 Q2 Q3 Q4
------- ---- ---- ---- ----
Finding Fido 1274 1671 1618 1605
Fly Fishing 716 918 1209 1248
...
```

The following UNPIVOT operation will rotate the quarter columns into rows. For each product, there will be four rows, one for each quarter.

```
SELECT *
FROM pivotedTable
UNPIVOT INCLUDE NULLS (quantity_sold
FOR quarter IN (Q1, Q2, Q3, Q4))
ORDER BY product, quarter;

PRODUCT QU QUANTITY_SOLD
------- -- -------------
256MB Memory Card Q1 1179
256MB Memory Card Q2 1533
256MB Memory Card Q3 1455
256MB Memory Card Q4 1374
...
64MB Memory Card Q1 414
64MB Memory Card Q2 215
64MB Memory Card Q3
64MB Memory Card Q4 81
...
```

Note the use of INCLUDE NULLS in this example. We can also use EXCLUDE NULLS, which is the default setting.

III. Materialized View Maintenance

Since materialized views correspond to pre-computed and stored query results, they may become out-of-date when the underlying sources are changed. Hence, one important issue is to maintain the materialized views' consistency upon any source changes. While re-computing views from scratch in response to any source updates may be acceptable for some relatively static databases, it is unaffordable when the source changes are frequent. Hence, incremental view maintenance, as an efficient alternative, has been proposed and extensively studied [5] – [7].

We can classify the existing view maintenance algorithms into two categories, namely, algorithmic and algebraic. Given a view and source updates, algorithmic view maintenance algorithms derive a program (a program can be a collection of deductive rules or SQL statements) whose evaluation maintains the view. The first proposal is the finite differencing algorithm, for incremental view maintenance under a functional data model. The output of the maintenance algorithm adds several lines of code into the source update transaction in order to also update the view. It assumes set semantics of all base tables and a key is required to exist in the view. A counting algorithm for maintaining views under bag semantics essentially keeps track of the multiplicity of each view tuple, or in other words, the number of derivations of each view tuple. The insert deltas have a positive count while the delete deltas have a negative count. A view tuple is deleted from the view if its count becomes 0.

The main issues with algorithmic view maintenance algorithms are that (1) the correctness of the algorithms is hard to prove, especially when the view language is extended, it is unclear and hard to prove if the existing algorithms will still work; (2) the output of maintenance algorithms (a program as mentioned above) is also hard to optimize. Hence algebraic solutions have been proposed to address these limitations. More specifically, an algebraic approach pre-defines a set of primitive change propagation rules for each operator. The maintenance plan can then be constructed by propagating changes through each algebra operator in the view query algebra tree and recursively applying those primitive rules. The output of such algorithms, namely, the maintenance plan, can be optimized by a cost-based query optimizer. Also since the algorithm is algebra-based, the result is not tied to any particular query language.

Due to these benefits mentioned above, algebraic view maintenance algorithms have been extensively explored. Most existing work builds upon such an algebraic maintenance framework by considering more types of operators or considering different underlying data models. Algebra-based maintenance work has also been studied beyond the relational data model, e.g., maintaining XQuery views based on an XML algebra. All the work above clearly reflects the extensibility of such an algebraic maintenance framework. Such extensibility lies in the fact that for each algebra operator, its change propagation is independent of its application context. Hence we can reuse the existing change propagation rules for the same operator in more complex language constructs.

Incremental view maintenance techniques are applicable to many other applications, such as trigger/constraint processing, cache/replica maintenance, to name a few. The view self-maintenance problem can also be considered as an application of the view maintenance techniques. That is, given a view and source updates, after generating the maintenance plan, we can easily determine if we need to query the sources or not. Some recent emerging applications, such as continuous query processing over data streams, are also closely related to the incremental view maintenance techniques.

IV. View Maintenance With Pivot And Unpivot

Relational database engines have been extended to natively support OLAP operations like drill down or rollup, slice and dice, pivot and unpivot in order to achieve better performance. Well-known examples include the extension of relational engine with CUBE and ROLLUP operators to support multidimensional aggregation, transposing operations implementation outside of RDBMS, implementations of pivoting and unpivoting through unfold and fold operations [8]-[10]. Fig. 2 shows the execution time of the PIVOT query and the equivalent sub-query formulation of the following queries.

```
SELECT year,
(SELECT sales FROM SalesTable t2 where month='Apr'
AND t2.year=t1.year) 'Apr',
(SELECT sales FROM SalesTable t2 where month='May'
AND t2.year=t1.year) 'May',
(SELECT sales FROM SalesTable t2 where month='Jun'
AND t2.year=t1.year) 'Jun',
FROM SalesTable t1 GROUP BY year;
SELECT * FROM
(SalesTable PIVOT (sales FOR month IN ('Apr', 'May',
'Jun')))
```

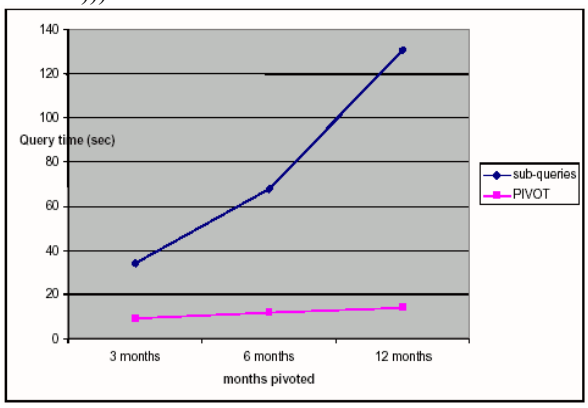

Fig. 2: PIVOT vs. scalar sub-queries

For each of the two forms, the number of months to PIVOT has been changed: three months of the year, then six months of the year and finally all twelve months. The performance difference is due to the duplication of the work in the sub-query information as each pivoted column is computed separately, because common sub-expression code does not

handle this case. More indices can be used to speed up the computation of the sub-query form, even if the common sub-expression is not detected. Fast lookup of the value from the dimension columns (e.g., index on year, month) would make performance comparable to the PIVOT form.

One view maintenance issue with PIVOT and UNPIVOT has been introduced in which the solution is based on the algebraic view maintenance [11]. This incremental refresh with PIVOT and UNPIVOT uses cost-based optimizer and query transformation. Here also proposed generalized PIVOT (GPIVOT) and generalized UNPIVOT (GUNPIVOT) for efficient incremental maintenance of views.

V. Research Trends In View Maintenance

We have described the PIVOT and UNPIVOT operators for OLAP operations in Data warehousing environment and materialized view which is the key issue for data warehouse. So the research works should focus on the maintenance of materialized view with these operators. Although there is a current solution of incremental maintenance of views with PIVOT and UNPIVOT and a proposal of new GPIVOT and GUNPIVOT but in this proposal the cost of generating maintenance plan has been ignored and stated that the cost can be minimum. So how to reduce the cost of generating maintenance plan can be a research topic for the view maintenance with PIVOT and UNPIVOT operators.

Most of the view maintenance techniques discussed on distributive functions and algebraic functions. The view maintenance with holistic functions such as median has still been unsolved and can be a research issue.

Maintenance of view with PIVOT and UNPIVOT operators based on distributive functions such as SUM, COUNT, on algebraic functions such as variance and regression, on holistic functions such as median are also some research issues to be solved in the near future.

VI. Conclusion

Materialized views in OLAP applications in the data warehousing environment are an important issue to get query response quickly.

View maintenance aims to incrementally maintain the view extent under a source data update. The idea is to issue a maintenance query based on the data update to calculate the delta change on the view extent and in this regard the operators PIVOT and UNPIVOT are playing important role for the commercial database as well as data warehouse. So the efficient maintenance of materialized view with PIVOT and UNPIVOT should give the focus to be solved in a perfect way. Here, we have described the query rewriting rules and view maintenance framework with these two operators to obtain an efficient view maintenance plan and pointed out the further research direction in this emerging field.

References

[1] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," SIGMOD Record, vol. 26, no. 1, pp. 65–74, 1997.

[2] C. Cunningham, C. A. Galindo-Legaria and G. Graefe, "PIVOT and UNPIVOT: optimization and execution strategies in an RDMS", Proceedings of the 30th VLDB Conference, Toronto, Canada, pp. 998-1009, 2004.

[3] S. R Valluri, S. Vadapalli and K. Karlapalem, "View relevance driven materialized view selection in data warehousing environment," Proceedings of the 13th Australasian Database Conference (ADC2002), Melbourne, Australia, pp. 187–196, 2002.

[4] Oracle® Database, *Data Warehousing Guide,* 11*g* Release 1 (11.1), B28313-02, September 2007.

[5] R. Agrawal, A. El Abbadi, A. Singh, and T. Yurek, "Efficient view maintenance at data warehouses," in Proceedings of SIGMOD, Arizona, USA, pp. 417-427, 1997.

[6] T. Griffin, and L. Libkin, "Incremental maintenance of views with duplicates," in Proceedings of SIGMOD, pp. 328-339, 1995.

[7] Y. Zhuge, H. Garc´ýa-Molina, J. Hammer and J. Widom, "View maintenance in a warehousing environment," in Proceedings of SIGMOD, pp. 316-327, May 1995.

[8] J. Gray, A. Bosworth, A. Layman and H. Pirahesh, "Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total," Proceedings of ICD, pp. 152–159, 1996.

[9] L. V. S. Lakshmanan, F. Sadri and S. N. Subramanian, "On efficiently implementing schema SQL on a SQL database system," Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, pp. 471-482.

[10] R. Agrawal, A. Somani, Y. Xu, "Storage and querying of ecommerce data," Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy, pp. 149-158.

[11] S. Chen, "Efficient incremental view maintenance for data warehousing," Ph. D dissertation, Worchester Polytechnic Institute, USA, 2005.