PRIORITY FREQUENCY MAPTABLE FOR QUERY OPTIMIZATION USING MATERIALIZED VIEWS

MD. RAFIQUL ISLAM, MORSHED U. CHOWDHURY
School of Information Technology, Deakin University, Victoria, Australia 3125
{rmd, muc} @deakin.edu.au

ABSTRACT

In this paper query optimization using materialized views has been analyzed and a comprehensive and efficient technique has been proposed to create Map-table. Materialized views can provide massive improvements in query processing time, especially for aggregation queries over large tables. To realize this potential, a number of existing techniques have been considered regarding the problem of maintaining materialized views as well as optimal searching time and memory overhead. Keeping this in mind, an optimal algorithm has been proposed in this paper for query optimization. It has been demonstrated that the proposed algorithm reduces the searching time substantially and reducing the memory size as well.

KEY WORDS

Materialized Views, MapTable, Query Optimization, Priority Frequency Table.

1.0 INTRODUCTION

When a view is defined, normally the database stores only the query defining the view. In contrast, a materialized view is a view whose contents are computed and stored. Materialized views constitute redundant data, in that their contents can be inferred from the view definition and the rest of the database contents. However it is much cheaper in many cases to read the contents of materialized view than to compute the contents of the view by executing the query defining the view. However a problem with materialized views is that they must be kept up-to-date when the data used in the view definition changes. Otherwise the materialized view becomes inconsistent. The task of keeping a materialized view up-to-date with the underlying data is known as view maintenance.

It may seem that materialized views should be used to evaluate a query whenever they are applicable. Also there are problems in the optimization of queries in the presence of a materialized view. In fact, blind applications of materialized views may result in significantly worse plans compared to alternative plans that do not use any materialized views. Whether the use of materialized views will result in a better or a worse plan depends on the query and the statistical properties of the database. Since queries are often generated using tools and since the statistical property of databases are time-varying, it should be the responsibility of the optimizer to consider the alternative execution plans and to make a cost-based decision whether or not to use materialized views to answer a given database. Such enumeration of the possible alternatives by the optimizer must be syntax independent and efficient. By syntax independent, we mean that the set of alternatives enumerated by the optimizer (and hence the choice of the optimal execution plan) should not depend on whether or not the query explicitly references materialized views. Thus the optimizer must be capable of considering the alternatives implied by materialized view. In particular, a materialized views may need to be considered even if the view is not directly applicable (i.e., there is no sub expressions in the query that syntactically matches the view). Also, more than one materialized views may be relevant for the given query. In such cases, the optimizer must avoid incorrect alternatives where mutually exclusive compatible views are used together while considering use of mutually compatible views.

The idea of query optimization using a materialized view is a new concept in the present research world. To speed up view matching, descriptions of every materialized view have been maintained in memory. To keep the description of a materialized view in memory, the concept of MapTable [1] was introduced. For keeping the description of every materialized view in memory, a data-structure called MapTable is implemented. MapTable keeps the information about queries equivalent to the given one.

Basically the MapTable is divided into two parts. The left part contains the name of the view and the right part contains the body of the view. For a given query we try to find any sub expression that matches with the left part of the MapTable. Then we just replace the matched sub expression with the right part of the MapTable that is connected to the name of the materialized view. That is we can use a materialized view just like a base table [6].

There is a traditional cost based query optimization algorithm called join enumeration algorithm, which is a simplification and abstraction of the algorithm

proposed by [9]. Using the algorithm of MapTable an extended algorithm for query optimization was proposed by [4]. This algorithm performs better than the [9]. However the existing MapTable creation algorithms [4,9] has some limitations. To eliminate the drawbacks of the existing algorithms, a new version of the algorithm of creating MapTable is proposed. The proposed algorithm not only optimizes the searching time but also reduces the memory overhead compare to any of the existing techniques.

2.0 THE CONCEPT OF MAPTABLE

Intuitively, each safe substitution [1] results in a new query, equivalent to the given one. We encode the equivalent queries by storing the information about safe substitutions in the MapTable data structure.

From the definition of safe substitution, it follows that every safe substitution of a query Q with respect to a rule L (x, y) ⟶ V (x) corresponds to a renaming σ for the rule. Therefore, we can encode the information about a safe substitution by the doublet [σ(L), σ(V)]. The first component in the doublet is called the delete list and the second component in the doublet is called the AddLiteral. The delete list denotes the sub expression in the query that is replaced due to the safe substitution σ and the AddLiteral denotes the literal that replaces delete list. Since L may have more than one literal, the delete list is a set of literals

However, AddLiteral is a single literal. The algorithm to construct the MapTable for a given query is shown in Algorithm **1**. The last for loop iterates over all literals in the query.

ALGORITHM 1: CREATING THE MAPTABLE

Procedure MakeMapTable(Q,*R*)

```
begin
  Initialize MapTable
  for each rewrite rule r : L.→ V in R do
     for each safe substitution σ from r to Q  do
         MapTable:= MapTable ∪ [σ(L), σ(V)]
    endfor
endfor
for each literal q ∈ Q do
       MapTable := MainTable ∪,[{ q},q]
  endfor
end
```

EXAMPLE 2.1

Consider the following one-level rewrite rule for Large-Dept

Dept (dno, size, loc), size >30→Large_Dept (dno,Loc)

We illustrate the enumeration of substitutions using three materialized views such as Large_Dept, Loc_Emp and Executive.

1) Consider the following query, which asks for employees who work at a department in SF.

Query(name): -Emp(name, doe, sat, age), size > 30
Dept (dno, size, SF)

It can be seen that the MapTable will have the following two doublets.

({Dept (dno, size, SF), size > 30},
Large-Dept(dno, SF)
({Emp(name,dno, sal, age), Dept(dno, size, SF)},
Loc_Emp(name, size, SF))

Observe that the doublets correspond to materialize views that are mutually exclusive.

2) Consider the query to find employees who earn more than 200000 and work in departments with more than 30 employees.

Q'(name) : -Emp(name, dno, sal, age) , sal > 200k,
Dept(dno, size, loc), size > 30

It can be seen that the MapTable will have the following two doublets which correspond to applications of mutually compatible materialized views.

({Emp(name, dno, sat, age) , sat > 200k},
Executive(name , dno, sal))
({Dept(dno,size, loc), size > 30},
Large_Dept (dno, loc))

Notice that these two doublets implicitly represent the alternatives to the given query.

3.0 LIMITATIONS OF THE EXISTING ALGORITHM

In this section, some limitations of the existing algorithm of creating a MapTable have been summarized. These are:

1. The main limitations of the concept of MapTable mentioned above is the storage problem. In this existing method, when a view is declared its entry is just appended to the MapTable. Therefore, that the size of the MapTable grows gradually as the number of views increases. When a view comes as an input there is no method of checking if the input view already exists in the MapTable or not. Therefore, duplicated data may be entered into the MapTable.

2. In this existing method there is no restriction on the size of the Maptable. Therefore it may keep some entries that are not necessary, thus wasting memory.
3. For a given query it tries to find any sub expression that matches with the left part of the MapTable, and then replacing the matched sub expression with the right part of the MapTable that is with the name of the materialized view. For this reason it has to search the entry of the MapTable. Here the search time is higher than the proposed algorithm.

4.0 THE PROPOSED ALGORITHM

To eliminate the drawback of the existing algorithm of creating MapTable, a new algorithm for creating the MapTable has been proposed based on "priority of frequency", i.e. the algorithm will count the frequency of each entry. It has been shown that the proposed algorithm for creating the MapTable will provide significant performance improvements over the previous algorithm for creating the MapTable. The main techniques of the proposed algorithm for creating the MapTable are as follows:

1. Keeping a frequency count for each entry in the MapTable.
2. For a given view, if it matches any entry in the existing MapTable, then we increase the frequency of the existing view by 1 and sort the MapTable.
3. For a given view, if it does not match any entry in the MapTable, and if there is enough free space within a given restriction, i.e. limit of MapTable, and then just append the view to the MapTable.
4. For a given view, if the MapTable is full, then replace the view with the lowest frequency by the given view.

ALGORITHM 2: PROPOSED ALGORITHM FOR CREATING THE MAPTABLE

Procedure MapTable (Q,R)

```
Begin
Input View ( Q )
Initialize MapTable
for each rewrite rule r : L→ V in R do
for each safe substitution σ from r to Q
        do {
          If ( Q matches with any View
              in the  MapTable )
           Then (Increase the Frequency of
              the existing View by 1 and Sort ( MapTable
              ) )
          Else If (There is a free space in the
              MapTable)
          Then (MapTable: = (MapTable ∪ [σ(L),
              σ(V) ] with frequency  1 )
      Else (Replace the View having lowest
    frequency with this new one)
          }
    end for
end
```

5.0 A COMPARATIVE STUDY

In proposed algorithm (Algorithm 2) of creating MapTable keeps frequency of each view in the MapTable. Also the MapTable always keep sorted by frequency. So, most of the necessary materialized view, that is the materialized view with higher frequency, will remain in the upper of the MapTable. Also another checking has been kept in the proposed algorithm of creating MapTable is that, if the input materialized view already existed in the MapTable then it will not append in the MapTable, rather just increase the frequency of the existing materialized view in the MapTable. So there is no chance of duplicity here. But in the case of the previous algorithm (Algorithm 1) of creating MapTable, does not keep any frequency if an input comes just append the input in the MapTable. So there may be duplicate data in the MapTable as well as data are not sorted. The algorithm is simulated and materialized view searching time is calculated. After simulation the following data are found and a graph is drawn. [Table 1, Figure 1]

In the existing method when a view is declared its entry is just append to the MapTable. Thais is why the size of the MapTable grows gradually as the number of views increases. Also it may insert duplicated data as there is no method of checking if the input view already exists in the MapTable. However in the case of the proposed algorithm this checking is done, so there is no chance of inserting duplicate data. Further more in the proposed algorithm, the size of the MapTable is fixed, based on the properties of the database. Let the size of the proposed algorithm for creating the MapTable be 15. Now the memory size of the MapTable will not increase beyond 15. Here a graph is presented for supporting the above discussion. [Table 2, Figure 2]

6. 0 DISCUSSIONS

A comprehensive approach for solving the problem of query optimization in the presence of Materialized views has been proposed. Materialized views may result in significantly worse plans compared to alternative plans that do not use any Materialized views. In this paper the materialized view has been proposed as like as base table. To keep the information of the materialized view a data-structure is encoded called Maptable. It has been analyzed using existing techniques and some limitations for creating Maptable have been found. Keeping this in mind, a new algorithm is proposed for creating the Maptable using "priority of frequency ". It has been demonstrated that the proposed algorithm performs better with respect to searching time, as well as memory overhead, compared to any of the existing techniques.

References

[1.] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, Kyuseok Shim "Optimizing Queries with Materialized Views." *ICDE 1995*: 190-200

[2.] Jonathan Goldstein and Per-Ake Larson. "Optimizing Queries Using Materialized Views: A Practical, Scalable Solution". *Microsoft Research, One Microsoft Way*, Redmond, WA 98052

[3.] Surajit Chaudhuri and Kyuseok Shim "Optimizing Queries with Aggregate Views" *Hewlett-Packard Laboratories*, 1501 Page Mill Road, Palo Alto, CA 943043 USA. IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA

[4.] Huyn, N **"**Speeding Up Materialized-View Maintenance Using Cheap Filters at the Warehouse." *Technical Report*, Surromed, Inc. 1999.

[5.] Hoshi Mistry , Prasan Roy , S. Sudarshan , Krithi Ramamritham "Materialized View Selection and Maintenance Using MultiQuery Optimization*" IIT-Bombay, Bell Labs, Univ. of Massachusetts-Amherst*, 2000.

[6.] P.O. Bunernan and E.K. Clemons "Efficiently monitoring relational databases" *ACM TODS*, 4(3):368-382, 1979.

[7.] J.A. Blakeley, P. A. Larson, and F. W. Tompa "Efficiently updating materialized views" *In Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 61-71, Washington, DC, May 1986.

[8.] U. S. Chakravarthy, J. Grant, and J. Minker "Logic-based approach to semantic query optimization" *ACM Transactions on Database Systems*, pages 162-207, June 1990.

[9.] T. Cormen, C. Leiserson, and R.L Rivest. "Introduction to Algorithms" *The MIT Press*, 1990.

[10.] A. Chandra and P.M. Merlin "Optimal implementation of conjunctive queries in relational databases" *In Proceedings of the 9th Symposium on Theory of Computing*, pages 77-90, New York, August 1977.

TABLE 1: COMPARISON OF SEARCHING TIME IN (MICROSECONDS) BETWEEN EXISTING SYSTEMS AND PROPOSED SYSTEM.

Number Of Materialized Views	Searching Time In (Microseconds) In Existing System	Searching Time in (μs) in Proposed System
100	22	15
200	115	74
300	183	127
400	239	175
500	318	230

FIGURE 1: SEARCHING TIME COMPARISON BETWEEN EXISTING AND PROPOSED SYSTEM**.**

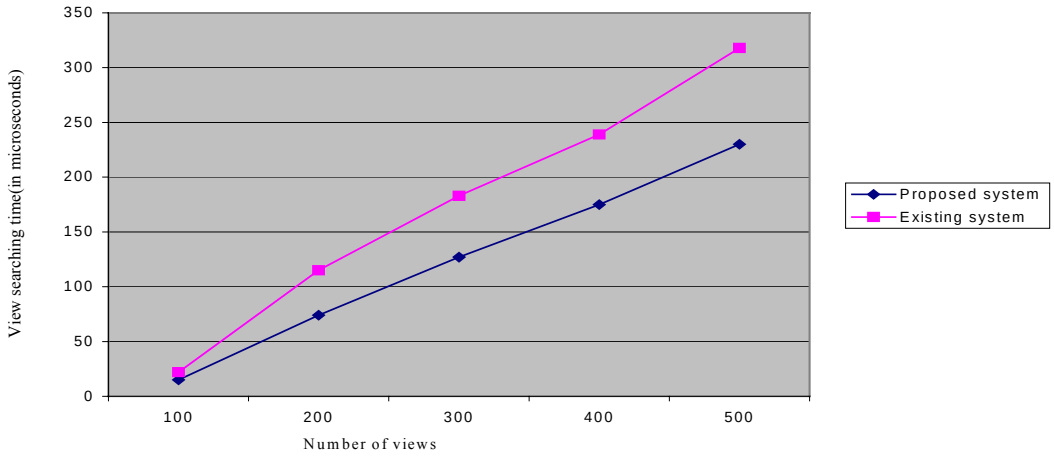

TABLE 2: MEMORY SIZE COMPARISON BETWEEN EXISTING SYSTEM AND PROPOSED SYSTEM.

Number Of Views	Memory Size For Existing System	Memory Size For Proposed System
100	5	4
200	10	8
300	15	13
400	20	15
500	25	15

FIGURE 2: MEMORY SIZE COMPARISON BETWEEN EXISTING AND PROPOSED SYSTEM**.**