Computer and Information Science; Vol. 5, No. 6; 2012
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

A Materialized View for the Same Generation Query in Deductive Databases

Nabil Arman[1]

[1] Department of Computer Science, Palestine Polytechnic University, Palestine

Correspondence: Nabil Arman, Department of Computer Science, Palestine Polytechnic University, Palestine. E-mail: narman@ppu.edu

Received: July 30, 2012 Accepted: August 24, 2012 Online Published: September 13, 2012

doi:10.5539/cis.v5n6p1 URL: http://dx.doi.org/10.5539/cis.v5n6p1

Abstract

Traditionally, deductive databases are designed as extensions to relational databases by either integrating a logic programming language, such as PROLOG, with a conventional relational database system that provides storage persistence needed for any database system, or by integrating an expert system with a relational database system. Deductive databases take advantage of a special kind of rule recursion called linear recursion to provide inference capabilities to improve the intelligence of the database system. The simplicity of implementation of linear recursive rules, like same generation rules, is far from the difficulty and the cost of computing the results of queries based on these recursive rules. Thus, to reduce costs and improve performance of the same generation queries, many techniques were suggested. In this paper, we propose the use of materialized views to speed up the evaluation of these queries and explain how to maintain the materialized view if the underlying base relation is updated. Finally, simulations are used to compare the materialized view approach with other approaches that are used to compute the results of the same generation queries.

Keywords: materialization, views, same generation query, deductive databases

1. Introduction

Deductive databases are designed as extensions to relational databases by either integrating a logic programming language, as PROLOG, with a conventional relational database system that provides storage persistence needed for any database system, or by integrating an expert system that provides the inferencing capabilities with a relational database system. Deductive databases and logic programming languages take advantage of a special kind of rule recursion called linear recursion since recursion improves the expressive power and intelligence of database systems (Arman, 2003). Therefore, efficient algorithms to process recursive rules and queries within the context of large deductive database systems has recently attracted a large amount of research efforts due to the important role of recursive rules in improving the intelligence of database systems and extending them into knowledge-base systems (Arman, 2004; Hopfner & Seipel, 2002; Elmasri & Navathe, 2010; Qadah et al., 1991). One of the main features of these intelligent database systems, namely deductive databases, is their ability to define recursive rules and to process queries on them directly.

Most recursive rules in deductive databases, appear in a simple form in which the rule's head appears only once in the body of the rule (Hopfner & Seipel, 2002). In general, this type of logic rules is called linearly recursive. A same generation (*sg*) rule is a linearly recursive rule of the following form:

sg(X,Y):-par(Z,X),par(W,Y),sg(Z,W)

where "*par*" is an extensional (base) predicate and "*sg*" is an intentional database predicate. Within the context of deductive databases, the extensional database predicate "*par*" is defined by a two-attribute normalized database relation with very many tuples as shown in Figure 1(a) (Arman, 2007). Another common view for the base relation is represented as a set of clauses in a PROLOG-like clause set as shown in Figure 1(b). A third common view of the base relation is represented by a directed acyclic graph, as shown in Figure 1(c).

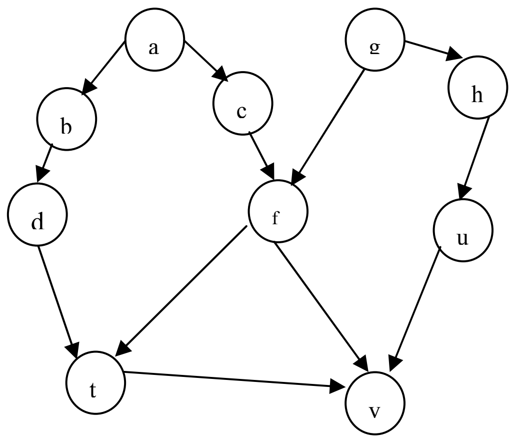
For every tuple <*x*,*y*> of the base relation, there exists, in the corresponding graph, a directed edge from node *x* to node *y*. The nodes in such a graph are the set of distinct values in the two columns of the base relation (i.e., the domain). In addition, there is a clause of the form *par(x,y)* that represents an assertion as it is written in predicate calculus.

X	Y
a	b
a	c
b	d
c	f
f	t
d	t
g	f
f	v
g	h
h	u
u	v
t	v

(a) Relation Form

```
par(a,b)
par(a,c).
par(b,d).
par(c,f).
par(f,t).
par(d,t).
par(g,f).
par(f,v).
par(g,h).
par(h,u).
par(u,v).
par(t,v).
```

(b) Clause Form



(c) Graph Form

Figure 1. The binary relation "par"

To generate solutions from the above recursive rule, another non-recursive rule, the trivial/exit rule, which defines the predicate "*sg(X,Y)*" must exist. This non-recursive rule is given by:

sg(X,Y):-par(Z,X),par(Z,Y)

A query on the predicate "*sg*" that is defined by the recursive and the exit rule is called a same generation query. This query is a headless rule of the following form:

:-sg(X,Y)

A query typically involves a predicate symbol with some variable arguments, and its meaning or answer is the different constant combinations that when bound (assigned) to the variables, can make the predicate true. A query such as the above may have different forms depending on the instantiation status of the variables (Ullman, 1989). If all variables are instantiated, the query is called fully instantiated and its result is either TRUE or FALSE. If only one variable is instantiated, the query is called partially instantiated and its result is the set of values of the uninstantiated variable that makes the query true. A query with no instantiated variables is called uninstantiated query and its result is the set of tuples <*x*,*y*> or the set of clauses *sg(X,Y)* of values that makes the query true.

Many techniques and algorithms were developed to answer queries based on the same generation rules. A special reverse matrix stores that stores paths from the nodes to the roots of the graph was used to answer queries by taking advantage of the paths ordering in the matrix structure. The algorithms developed outperforms other techniques like the counting technique for linear rules and the magic-sets rule rewriting, which are the two best-known techniques to solve such query in its simplest form (Arman, 2004; Arman, 2005; Banchilson et al., 1986; Suzuki et al., 1993). Parallel algorithms were also developed to answer different forms of the same generation query including the generalized same generation forms (Arman, 2006). Materialized views were used with transitive closures in deductive databases and proved to be very beneficial in improving the performance of this query (Toroslu, 1998). In this paper, materialized views are used to speed up the performance of the same generation queries, which are more costly than transitive closure queries and occur frequently in deductive databases and logic programming.

2. Materialized Views for Same Generation Query

A view is a derived relation from a set of base relations. Generally, views are not stored but are realized when queries based on them are executed. A materialized view is a view whose tuples are stored in the database and thus retrieved instantly when needed. Therefore, there are two main issues in view materialization. The first issue has to do with the definition of the view and its storage in the database and the second issue has to do with how to maintain the contents of the materialized view when updates are performed on the base relations. Thus, the definition of the materialized view using the "*par*" base relation on which the same generation query is defined has to be implemented and the issue that has to do with how to maintain the materialized view if the underlying "*par*" base relation is updated should be handled.

To generate all tuples of the same generation query, the uninstantiated form of the query is computed. This is a costly step but it is considered as a preprocessing step that is needed to answer all relevant queries after words. We used a PROLOG-like inference engine to generate all these tuples. To do so, a new predicate called the materialized same generation "*msg(X,Y)*" is used. "*msg(X,Y)*" can be defined and obtained using the same generation rules of the form:

sg(X,Y):- par(Z,X), par(Z,Y), assert(msg(X,Y)).

sg(X,Y):- par(Z,X), par(W,Y), sg(Z,W), assert(msg(X,Y)).

In the rules, the function *assert()* is used to add the tuples to a database that can be defined as:

```
database
    msg(symbol,symbol)
```

Thus, a PROLOG-like query of the form:

?sg(X,Y)

generates all tuples of the nodes that are of the same generation and the *assert()* function inserts them in the database as shown in Figure 2. The whole database can be saved in a text file which can be stored in a relational database. Since some tuples may be generated more than once, depending on the many paths that may exist between nodes of the same generation, if a tuple already exists in the database, it is not inserted again. The checking for that can be performed easily depending on an index structure that is built using the nodes of the tupels in the materialized view of the same generation query/rule.

The procedure for materializing the view of the same generation query/rule using the base relation *"par"* can be presented as:

```
MaterializeSameGeneratoin(Input: base relation "par", Output: materialized view "msg")
{
Read base relation "par"
Assert its tuples as "par" clauses in a PROLOG-like engine
Use a PROLOG-like engine to generate all tuples <X,Y> of materialized view "msg"
Using an index of tuples on <X,Y>, if tuple <X,Y> is not in "msg", store tuple <X,Y> in "msg"
}
```

msg

X	Y
b	c
c	b
t	v
f	h
v	t
h	f
d	f
f	d
t	u
v	u
u	t
u	v

Figure 2. The Materialized view "msg"

To answer a same generation query like ?$sg(c,W)$, the materialized view is used and a simple relational algebra query like $\pi_W(\sigma_{X=c}(msg))$ can be used and the result is computed instantly.

The view materialization of the same generation query results is a pre-processing step that can be computed off-line. Thus, the computational complexity or the time needed for such a step is not important in our case.

View maintenance is needed to make sure that the materialized view reflects the database in its current state. When the original base relation ”*par*” is modified with an insertion like “insert into par values(a,b)” on the “*par*” base relation, then the materialized view is modified by asserting the result of “*sg(x,a)*” and the result of “*sg(x,b)*” to the materialized view *“msg”*. Since “*sg(X,Y)*” is a symmetric relation, there is no difference between “*sg(x,a)*” and “*sg(a, x)*” or between “*sg(x,b)*” and “*sg(b, x)*” Thus, the modification of the materialized view is performed in such a way that the updates should not require the re-computation of the materialized same generation view “*msg*”.

However, deletions of existing tuples from the base relation “*par*” are handled by re-computing the materialized view of the same generation of the graph portion that is connected to the nodes of the deleted tuple (*x,y*) form the base relation “*par*”. This means that deletions may affect just the portion of the graph that is connected to the nodes in the tuple and some paths in the underlying graph of the base relation “*par*” and thus partial results of the same generation queries. Therefore, the re-computation of the whole materialized view of the same generation is never needed.

3. Results and Performance Evaluation

To determine the performance of the materialized view approach, several simulations were performed for random database relations with 1000 tuples of 4 different outdegree values from 1 to 4 of the underlying graph representing the base relation. The simulations were performed on an HP Compaq PC with Intel Core 2 Duo E4400 Processor (2.00-GHz, 2 MB L2 cache, 800-MHz FSB) running MS Windows 8 operating system. The materialized view approach of the same generation query was tested for 200 randomly generated queries (fully instantiated and partially instantiated). The same 200 randomly generated queries were answered using the traditional same generation query approaches. The time taken to answer these queries was determined for both techniques and the times were plotted for different outdegrees as shown in Figure 3.

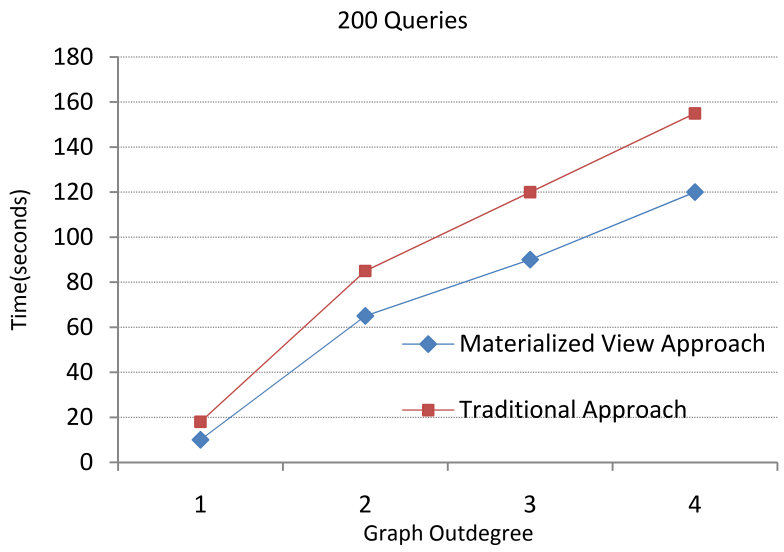


Figure 3. Performance for materialized view vs. traditional same generation query approaches

As shown in Figure 3, the materialized view approach for answering the same generation queries outperforms the traditional approach. Since these simulations are based on a database, the dominant factor is the time needed to compute the query results, rather than the main memory needed.

The view materialization of the same generation query results is highly needed when there is a large database that includes a relation like a *“par”* base relation on which same generation rules and queries can be defined and the underlying graph representing such a database has a large outdegree. If the database is relatively small or the outdegree of the underlying graph is small, there will be no justification for the materialization of the same generation query results.

4. Discussion and Conclusion

In this paper, the use of materialized views to speed up the evaluation of the same generation queries in deductive databases is proposed. This approach makes use of the computed materialized view that results from asserting all tuples of nodes of the same generation using a PROLOG-like inference engine. The same generation queries can then be computed instantly based on the materialized view. In addition, the materialized view approach was compared with traditional approaches for answering the same generation queries and the simulations demonstrate that the materialized view outperforms the traditional approaches.

References

Arman, N. (2003). An Intelligent Algorithm for the Generalized Fully Instantiated Same Generation Query in Deductive Databases. *Proceedings of the 4th International Arab Conference on Information Technology (ACIT'2003)*, pp. 224-228, December 20-23, Arab Academy for Science and Technology, Alexandria, Egypt.

Arman, N. (2004). An Efficient Algorithm for the Generalized Partially Instantiated Same Generation Query in Deductive Databases. *The International Arab Journal of Information Technology, 1*(1), 142-146.

Arman, N. (2005a). Graph Representation Comparative Study. *Proceedings of the 2005 International Conference on Foundations of Computer Science (FCS'05)*, June 27-30, Las Vegas, USA.

Arman, N. (2005b). Graph Representation: Comparative Study and Performance Evaluation. *Information Technology Journal, 4*(4), 465-468. http://dx.doi.org/10.3923/itj.2005.465.468

Arman, N. (2006a). Parallel Algorithms for the Generalized Same Generation Query in Deductive Databases. *Journal of Digital Information Management, 4*(3), 192-196.

Arman, N. (2006b). A Parallel Algorithm for the Generalized Fully Instantiated Same Generation Query in Deductive Databases. *Proceedings of the 4th International Multiconference on Computer Science and Information Technology (CSIT2006)*, vol. 1, pp. 282-288, April 5-7, Applied Science University, Amman, Jordan.

Arman, N. (2006c). A Parallel Algorithm for the Generalized Partially Instantiated Same Generation Query in Deductive Databases. *Proceedings of the 2006 International Conference on Information and Knowledge Engineering (IKE'06)*, June 26-29, Las Vegas, USA.

Banchilon, F., Maire, D., Sagiv, Y., & Ullman, J. (1986). Magic Sets and other Strange Ways to Implement Logic Programs. *Proc. 5th ACM Symp. On Principles of Database System*, pp. 1-15.

Elmasri, R., & Navathe, S. (2010). *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc.

Hopfner, M., & Seipel, D. (2002). Reasoning about Rules in Deductive Databases. *Proc. 17th Workshop on Logic Programming (WLP'2002)*.

Qadah, G., Henschen, L., & Kim, J. (1991). Efficient Algorithms for the Instantiated Transitive Closure Queries. *IEEE Transactions on Software Engineering, 17*(3), 296-309. http://dx.doi.org/10.1109/32.75418

Suzuki, S., Kishi, M., & Ibaraki, T. (1993). Query Evaluation of the Same Generation Problem with Many Variables. *Systems and Computers in Japan, 24*(10), 1-14.

Toroslu, I. (1998). View Maintenance for Materialized Transitive Closure Relations. *Journal of Database Management, 9*(1), 3-12.

Ullman, J. (1989). *Principles of Database and Knowledge-Base Systems.* Computer Science Press.