Simulation Results for a Cache Management System used in a Deductive Database

Larry Williams¹, Martin Maskarinec¹, and Kathleen Neumann¹
¹School of Computer Sciences, Western Illinois University, Macomb, IL

Abstract - This paper will present the effectiveness of a cache management system which was previously developed for a Deductive Database. In order to do this, a simulation of user inputted queries was developed and then executed on two different input sets. This paper presents the results of these simulation runs and a comparison between the results of our caching algorithm as compared to two other, standard approaches.

Keywords: Intelligent Database, Deductive Database, Cache Algorithms

1. Introduction

A deductive rule uses predicates to represent knowledge that may be derived from known facts. For example, we may write “P(X,Z) :- F1(X,Y), F2(Y,Z)”. This rule indicates that the predicate “P” is dependent on the facts “F1” and “F2”. These predicates and facts may have arguments; the variables X, Y, and Z indicate how the arguments of the resulting predicate P are derived from the predicates of F1 and F2 and any other constraints (in this example, the second argument of F1 must match the first argument of F2).

A deductive database is a collection of these rules. These rules may contain many facts and/or other predicates. As the set of rules expands, it is convenient to view it as a “Predicate Connection Graph”, such as the one depicted in Figure 1. This particular Connection Graph shows a rather large number of predicates and facts; a user may query any node in the Graph at any time. See [2] for a complete discussion of how queries are processed in such a system.

As more and more queries are presented to the deductive database on larger and larger predicate results, it becomes important to cache previously realized results so, if they are queried again, the results are readily available and do not need to be recalculated. As with all caching algorithms, it is unreasonable to assume all previously realized results may be saved in memory – there is just too much data for too many nodes. Thus, the important question to answer here is: when the maximum amount of available memory is exceeded, what should be saved to the cache and what should be removed and left to be recalculated when queried?

There are several caching algorithms available for relational databases (see [3] for an example). However, these do not take into account the dependencies of predicate caches. As can be seen in Figure 1, removing the cache for node P2 has two major ramifications: 1) most obviously, nodes down the tree would need to be referenced to recreate the result for P2 if it is queried again, and 2) nodes up the tree would also become more expensive to recalculate, since they reference P2 and the cost of acquiring the result for P2 will go up if its cache is removed.

Previously, we created a caching algorithm to take into account these special requirements of a deductive database [1]. In this paper, we present our results of testing this algorithm to determine just how effective it is.

2. Review of Caching Algorithm

When a query request comes in to the Deductive Database, the system will first check to see if it is cached. If so, a hit counter will be incremented and the results returned. If not, the results will be calculated by querying the children nodes for their results and then putting these results together as required by the predicate rule. Once this result is compiled, it needs to be determined if it should be cached or not. This is done by determining the node’s “Recalculation Cost”.

The recalculation cost of a node is a function of the cost of producing the result of the node, the number of parents, and the miss percentage. Briefly, the cost of producing the result of the node looks “down” the predicate graph, while the number of parents looks “up” the predicate graph. This miss percentage is also factored in since a large number of misses would indicate a frequently queried node that is not currently being cached. See [1] for a complete description of the recalculation cost methodology.
3 Simulation

The simulation compares three different types of cache algorithms: First In, First Out; Last In First Out; and the Recalculation Cost algorithm described in [1]. For each of the three algorithms, the same predicate graph will be read from a file. This file will contain the list of predicates and their connections; also, simulated size information will be provided as well. Then, a set of 100 simulated queries will be put to each algorithm to simulate user input. These queries will also be read from a file to ensure all algorithms are given exactly the same input.

We have created two sets of 100 simulated queries. The first set is one that contains relatively few repeated queries (we refer to this one as the “low performance” set because the cache will not be hit very often in any case). The second set contains relatively many repeated queries (we refer to this one as the “high performance” set because the cache algorithm should enable many hits). We will then measure system performance for both of these sets in terms of hit percentage as well as number of bytes retrieved from disk (for cache misses).

Next, we describe the three algorithms in more detail.

3.1 First In, First Out

In the First In, First Out algorithm, the system will cache all the nodes until the cache is full; once the cache is full, the cache management system will remove the node that was placed in the cache first. This algorithm acts basically like a queue.

3.2 Last In, First Out

In the Last In, First Out algorithm, the system will cache all the nodes until the cache is full; once the cache is full, the system will remove the last node that was entered into the cache. This algorithm acts basically like a stack.

3.3 Recalculation Cost Algorithm

In the Recalculation Cost Algorithm, the system will cache the nodes based on the algorithm described in [1].

4. Results

Figure 2 shows the hit rate for all cases. As can be easily seen, the FIFO algorithm running on the high performance set trends much worse than the other two as more and more hits occur. The LIFO algorithm and the recalculation algorithm trend very similarly with the recalculation algorithm slightly out-performing LIFO by the end of the simulation. On the low performance set, all three trend similarly.
At first glance, this seems to indicate the recalculation algorithm is not much better than the LIFO algorithm. However, it must be remembered that Figure 2 shows hit rate and this is not what the recalculation algorithm is designed to optimize. A large number of hits on small and/or easily recalculated predicates may increase the hit percentage, but caching larger and/or more expensive predicates to recalculate would save time.

To demonstrate this, Figure 3 presents the same simulation runs, but looking at total bytes read from disk, rather than a simple hit rate. Thus, each time a miss occurs, the number of bytes necessary to be retrieved to construct the result is calculated and added to a running sum. The final values for this running sum are shown in Figure 3 for all three algorithms run on both input data sets.

As can be seen in Figure 3, the recalculation method clearly outperforms the other two for both input data sets. In fact, it requires only about 60% of the disk accesses of the other two cases in both input data sets.

5. Conclusion

A simulation of a cache management system has presented which demonstrates this system outperforms basic FIFO and LIFO techniques. Using our recalculation cost method, 50% less disk accesses are required to meet the simulated user’s requests.

6. Bibliography

