Database Query Optimization Workshop, Portland, May 1989.

QUERY PROCESSING FOR DATABASES WITH PROCEDURAL VALUES

Yao-Nan Lien and Shu-Shang Wei

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277
(614) 292-5236,{lien, wei-s}@tut.cis.ohio-state.edu

ABSTRACT

This project is investigating the performance problem of databases with procedural values and proposing a new approach to process queries for such databases. To process a query in such databases may induce many secondary queries if the query includes procedure attributes in its predicate as variables. To minimize I/O overhead in processing such a query, we follow the same direction as Sellis' to process more than one induced query at the same time, but with a different approach, the *predicate-cascading (PC)*. The PC approach will reduce the I/O overhead by having each data unit brought into the internal memory to be evaluated against all applicable secondary queries simultaneously. To apply this approach, it is neither necessary to have any implication relationship among queries nor to invoke a complicated searching process to determine the best query plan. With appropriate concurrency control, this approach can be applied to the general multiple query optimization problem as well. A benchmarking experiment is being conducted in the Ohio State University to evaluate the proposed approach when it is applied to multiple query optimization.

1. INTRODUCTION

Traditional relational database systems are primarily designed for business applications. The use of a table-like data model and non-procedural query languages greatly simplifies the management of databases. After being recognized as a dominating model, the relational data model is being introduced to other application areas, such as knowledge base and engineering design automation. Many new extensions are being proposed to extend the descriptive power of relational model. Among all newly proposed extensions, the *procedural-value relational database* (procedural-value database), which accepts procedures as primitive values, is an attractive approach.[8] An example consisting of four relations in INGRES+ is as follows:

EMP (name, age, salary, hobbies)
SOFTBALL (emp-name, hour-spent, position, average)
SAILING (emp-name, hour-spent, rating, boat-type, marina)
JOGGING (emp-name, hour-spent, distance, best-time, number-of-races).

The attribute "hobbies" in relation EMP is a procedure attribute possibly with the following form:

retrieve (REL.all) **where** REL.emp-name = "value-for-this-employee",

where REL is one of the hobby relations.

To process a query in such a database may induce many secondary queries if the query needs to retrieve or to evaluate any procedure attribute (EMP.hobbies in the above example). The processing overhead will be significantly higher than that in a traditional relational database, especially when the database is disk based, where the system performance is very sensitive to the I/O overhead. After a careful evaluation of the evolution of the computer technology, we find that the speed mismatch between the processor and the I/O is likely to be enlarged, not reduced. Therefore, the procedural-value

database may not be able to offer a satisfactory performance even for simple queries unless the I/O overhead can be reduced.[6]

2. MULTIPLE QUERY OPTIMIZATION

It is appropriate to apply the so called *multiple query optimization* technique to this database.[7] The basis of this approach is to treat the induced secondary queries as a set of independent queries and to optimize the whole set of queries together using the multiple query optimization technique.[3,2,5,1,4,6]

In multiple query optimization, the *common subexpression reduction* is the most popular approach. It identifies the implication relationship among queries (i.e. the result of one query covers the result of the other) and pipelines the result of one query to another. Then a combinatorial search procedure is invoked to determine the best execution plan to maximize the overhead reduction.

To apply the common subexpression reduction to procedural-value databases may not be very effective. First, the number of tuples returned by a secondary procedures in a procedural-value database is normally very small so that it may not exist a large overlap among secondary queries. Secondly, a large number of secondary queries may be induced by a given query so that the best execution plan can not be determined in a reasonable amount of time. Therefore, there is a need to develop a better approach that does not rely on the implication relationship and does not need a combinatorial search.

3. PREDICATE CASCADING APPROACH

Judging from the fact that I/O is the major bottleneck of large database systems, we propose a new approach, the *predicate-cascading (PC),* to reduce the I/O overhead without encountering the problems mentioned above. This approach evaluates each data unit brought into the internal memory against all applicable secondary queries together. At the extreme case, the same data object needs to be read only once for all procedures that reference the same relation. To apply this approach, it is not necessary to have any implication relationship among queries nor to invoke a complicated searching process to determine the best query plan. With an appropriate concurrency control, this approach can be applied to the general multiple query optimization problem as well.

The PC approach uses a three phase query processing strategy to process an incoming transaction. In the first phase, the *syntactic reduction phase,* all procedures contained in a procedure attribute are analyzed and partitioned into disjoined *procedure groups* according to the relations they reference. In the second phase, the *secondary qualification phase,* all secondary procedures are processed group by group; relations referenced by each procedure group are read; and then all procedures in the group are processed using predicate-cascading operations. In the third phase, the *ordinary qualification phase,* the ordinary predicate is evaluated. The second phase is actually converting the secondary procedures in procedure attributes into atomic values so that the system can process it as a traditional relational database in the third phase. For each page read from the secondary storage into the internal memory, the query processor retrieves the tuples in this page that are qualified for the first procedure in the group and then retrieves the tuples that are qualified for the next procedure immediately before next page is read. This research effort is to reduce the overhead in the secondary qualification phase.

In the rest of this section, we briefly describe the optimization of simple selections and simple two-way joins, where secondary procedures do not include procedure attributes. They are referred to as *PC-selection* and *PC-join,* respectively. A multiway join is decomposed into a sequence of two-way joins and processed by PC-joins.

3.1. PC-selection

PC-selection is used to process a group of secondary selections that reference the same relation regardless of what the predicates are even if the variables are in different attributes. The secondary procedures in this case have the following simplest format:

retrieve (secR.*)
where secR.secF *op* constant

Where "secR" is the secondary relation referenced by the procedure in the group; "op" is the comparison operator; and "secF" is the attribute to be compared with "constant". All procedures that are grouped into the same procedure group need to reference the same secR. There is no restriction imposed on "secF", "op", or "constant". Although an actual predicate may consist of more than one term, it will not make the problem more difficult.

During the syntactic reduction phase, a predicate table is generated for each procedure group. The table is usually small enough to reside in the internal memory when the procedure group is being processed. A buffer, called *selection-collater* (collater, in short), is associated with each procedure to collect the results. Whenever a page is read into the internal memory, all tuples in the page are evaluated against all procedures in the group before next page is read. The results are sent to the corresponding collater. If a collater is too small to store all possible results, a larger buffer in the external storage will be created to expand the in-memory collater. Since the external buffer may significantly increase the I/O overhead, it is necessary to develop some way to minimize the need of external collater. The followings are some examples to do so:

1. Pipeline the result generated by a procedure to the ordinary qualification immediately before the data is evaluated against the next procedure.
2. Create an index table to store the pointers of each tuple and a set of boolean vectors, one for each secondary procedure. Each bit in a vector corresponds to the qualification of a tuple in the targeted relation when it is evaluated against the corresponding procedure. Whenever a tuple is qualified for a procedure, the corresponding bit is set in the vector. The index table together with these boolean vectors can be used in the ordinary qualification phase to retrieve the tuples from secondary relations.

3.2. PC-join

If a secondary procedure references more than one relation, a join operation will have to be invoked to instantiate the "procedural-value" for the ordinary predicate. It is well known that a join operation may induce a much higher overhead than a selection. We follow the same approach that the PC-selection takes to process all two-way joins that reference the same pair of relations together. In the syntactic reduction phase, all procedures that need to join the same pair of relations are grouped into the same procedure group. The secondary procedures in this case have the following format:

retrieve (secR1.*, secR2.*)
where secR1.secF *jop* secR2.secF
and secR1.secF1 *op*1 constant1
and secR2.secF2 *op*2 constant2

The first term in the predicate is a join operation that joins relations secR1 and secR2 over the joining attribute secF on the join operator *jop*. The predicate "secR1.secF *jop* secR2.secF" is referred to as the *join predicate*. The second term and the third term are selections applied on secR1 and secR2,

respectively.

Unlike conventional query processors where the two selections may be executed before the join in order to reduce the joining overhead, the selections in PC-join will not be executed before the join. Since only those procedures that use the same join predicate can share the same join result, the procedures in the same procedure group are further partitioned into smaller groups according to their join predicates. Two procedures are in the same subgroup only if their joining attributes and joining operators are identical. The results of a particular join can then be used by all procedures in the same subgroup for further processings. (Note that the partitioning can be done in the syntactic reduction phase.) The PC-join itself is divided into two phases. All joins are processed in the first phase and all selections are then applied to the outputs of join operations in the second phase. In the first phase, a join processor and a join-collater are set up for each individual join. Without loss of generality, the nested-loop join is used and one tuple from secR1 and one tuple from secR2 are joined in each step. As shown in Figure 2, whenever a pair of tuples from both relations is fetched, it is sent to all join processors. The output of a join processor is saved in the associated join-collater. In the second phase, all selections of each subgroup are applied using PC-selection to the tuples in the corresponding join-collater to get the final results for all secondary procedures. Similar to the PC-selection, a data compression and a memory management scheme are needed to reduce the storage requirement of PC-join.

4. SUMMARY

Following are some of the problems that remain to be solved at this stage.

1. The data compression and memory management for various collaters.
2. The decomposition of multiway join and ways to reduce the overhead when the same data object is needed in different joining steps.
3. Index management and the query optimization when indices are available.
4. Recursive procedural value evaluation, i.e. a secondary procedure also needs to evaluate procedural values.
5. The extension of PC-approach to other extensible databases.

Because there is a significant increase in the ratio of processor overhead and I/O overhead (it is possible that the processor overhead overruns I/O overhead in PC approach), a system with a larger processor capacity may be needed to balance the I/O load and processor load. With its cost-effectness, the multiprocessor system would be a good candidate to support the PC approach. It is not difficult to see that there exists a high degree of parallelism in this approach so that an efficient parallel execution of this approach can be easily developed. However, further research has to be done before any conclusion can be drawn.

Of course, the final conclusion of the PC approach cannot be made without an appropriate performance evaluation. A benchmarking experiment is being conducted at the Ohio State University to evaluate the PC approach when it is applied to the multiple query optimization and the results will be reported in a forthcoming paper.

References

1. Chakravarthy, U. S. and J. Minker, "Processing multiple queries in database systems," *Database Eng.*, vol. 5, No. 3, pp. 38-44, Sep 1982.

2. Grant, J. and J. Minker, "Optimization in deductive and conventional relational database systems," *Advances in Data Base Theory*, vol. 1, pp. 195-234, Plenum Press, New York, 1981.

3. Hall, P. V., "Common subexpression identification in general algebraic systems," *Tech. Rep. UKSC 0060*, IBM United Kingdom Scientific Centre, Nov. 1974, Nov. 1974.

4. Kim, W., "Global optimization of relational queries: a first step," *Query Processing in Database Systems*, pp. 206-216, Springer-Verlag, New York, 1984.

5. Roussopoulos, N., "View indexing in relational databases," *ACM Trans. Database Syst.*, vol. 7, No. 2, pp. 258-290, June 1982.

6. Sellis, Timos K. and Leonard Shapiro, "Optimization of Extended Database Query Languages," *Proc. ACM-SIGMOD*, pp. 424-436, May 1985.

7. Sellis, Timos K., "Multiple-Query Optimization," *ACM Trans. on Database Systems*, vol. 13, No. 1, pp. 23-52, March 1988.

8. Stonebraker, Michael, Jeff Anton, and Eric Hanson, "Extending a Database System with Procedures," *ACM Trans. on Database Systems*, vol. 12, No. 3, pp. 350-376, Sep. 1987.

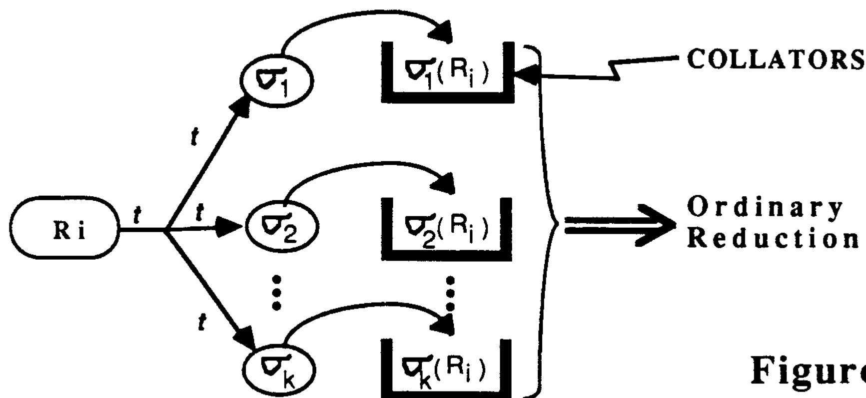

Figure 1. PC-selection

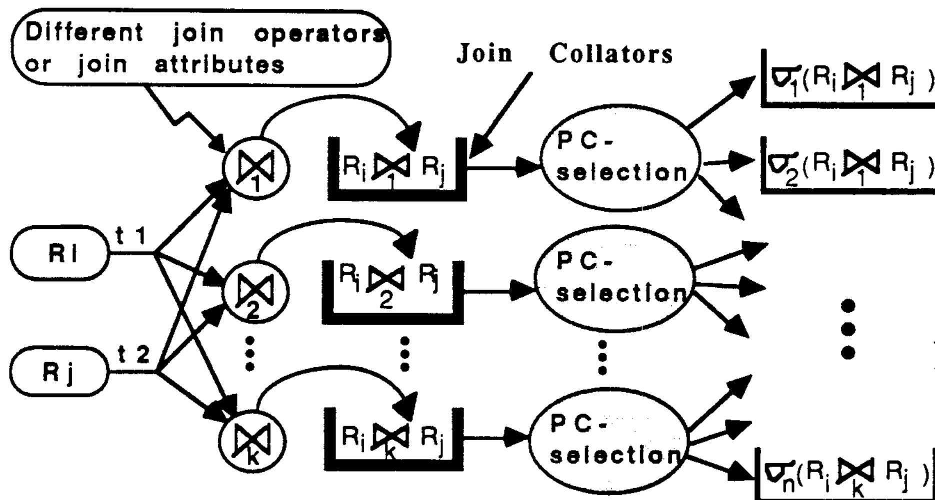

Figure 2. PC-join