Database Query Optimization, . Proceedings of the ODBF Workshop

Goetz Graefe

Technical Report No. CS/E 89-005

May 1989

Preface

I appreciate this opportunity to bring researchers and practitioners together for this workshop on database query optimization. This topic has been of great importance for relational database systems, and it will be central for future computer systems.

In a narrow scope, the performance of database systems can be improved through work in two areas, query planning and query processing. The most significant challenges in these areas, in my opinion, are queries and operations over complex objects and effective use of parallelism.

In a broader scope, we will continue to see a development towards non-procedural computer languages. Because a non-procedural language requires less user input than a procedural language, such languages are more attractive. However, if the user does not specify execution steps, the computer system must include a planning component. Carrying the technology of database query optimization into other areas of computer science and broadening the scope of execution planning and optimization will make a real difference in future computer systems.

I would like to extend my appreciation to all authors who submitted extended abstracts to this workshop. I received 38 extended abstracts, including 3 from Canada and 5 from Europe. The depth and width of the work are very exciting; I hope that this depth will be reflected in the discussions, and will be a valuable experience for all of us.

I want to thank Stavros Christodoulakis, Yannis Ioannidis, and Guy Lohman for preparing and moderating the discussion sessions during the workshop. Guy Lohman also read all submissions and made many insightful comments. Kelly Atkinson and Kathy Hammerstrom of OGC provided the all-important administrative support. Finally, my thanks to the Oregon Database Forum, the Oregon Center for Advanced Technlogy Education, and ACM SIGMOD, in particular David Maier and Roger Olson, for their sponsorship.

Goetz Graefe
May 1989

Table of Contents

Graefe, G. 1
Research Problems in Database Query Optimization

Lohman, G. M. 13
Is Query Optimization a "Solved" Problem?

Selinger, P. G. 19
Five hard problems in query optimization

Rosenthal, A. 23
Discussion Issues for Query Processing

Krishnamurthy, R. 29
Research directions in the Optimization of a Logic Based Language

Ramakrishnan, R. 37
Two Problems in Recursive Query Optimization

Freytag, J. C. 41
Increasing the Flexibility of Query Optimization

Siegel, M. D. 47
Heuristic-Based Semantic Query Optimization and Automatic Rule Derivation

van Kuijk, H. J. A. and Apers, P. M. G. 53
Semantic Query Optimization in Distributed Databases: A Knowledge-Based Approach

Yoon, S. and Henschen, L. J. 59
Intelligent query answering in Expert Database Systems using a heuristic approach

Heiler, S. 65
An Example of Semantic Query Optimization as a Technique for Improving Query Performance in Federated Systems

Pirahesh, H. 71
Early experience with rule-based query rewrite optimization

Dayal, U. 77
Distributed Object-Oriented Query Optimization: Perspectives, Problems, and Paradigms

Hasan, W., Lyngbaek, P., and Wilkinson, K. 85
Expression Processing in Iris

Osborn, S. L. 91
Query Optimization Basics for Object-Oriented Databases

Shaw, G. M. and Zdonik, S. B. 95
Efficient Querying in an Object-Oriented Database

Vandenberg, S. 101
Practical Complex Object Algebras

Moss, J. E. B. 109
Using Object-Oriented Subtyping in Query Optimization and Processing

Pathak, G. and Blakeley, J. A. 115
Query Optimization in Object-Oriented Databases

Scholl, M. H. 119
On the Complexity of Nested Relational Operations

Blakeley, J. A. and Deshpande, A. 125
Query Processing in a Nested Relational Database System

Ioannidis, Y. E. and Kang, Y. 131
Randomized Algorithms For Optimizing Large Join Queries

Swami, A. 139
Research in Optimization of Large Join Queries

Chang, P. 143
On The Optimization Of Large Equi-Join Queries

Yoo, H. and Lafortune, S. 149
Heuristic Search in Query Optimization

Chakravarthy, S. 153
Multiple Query Optimization: To Use or Not To Use?

Segev, A. and Gunadhi, H. 159
Query Optimization In Temporal Databases

Chakravarthy, S. 165
Optimization of Alerters/Triggers/Rules in Active DBMSs

Lien, Y. and Wei, S. 169
Query Processing For Databases With Procedural Values

Muralikrishna, M. 175
Optimization of Nested Tree Queries

Hammond, B. 181
BLAST Query Optimization

von Bültzingsloewen, G. 189
Optimizing SQL Queries for Parallel Execution

Luk, W. S. and Wang, X. 197
Communication Considerations for Query Optimization for Distributed/Parallel Systems

Martin, T. P. 203
Site Selection in Distributed Query Processing

Whang, K. and Krishnamurthy, R. 209
Query Optimization in a Main-Memory-Resident Database System

Weddell, G. E. 217
Query Optimization for Memory-Resident Databases

Kirkpatrick, J. E. and Roth, M. A. 225
A Performance Study of Nested Relational DBMSs involving Query Optimization

Tong, L. and Roussopoulos, N. 231
An Operational Optimization Approach for Parallel n-join with Large Number of Processors

L. Raschid 235
Evaluation of Linear Recursive Predicates in Deductive Databases

L. Raschid 237
Query Compilation and Rule Storage in a Knowledge Base Management System

Author Index

Apers, P. M. G.	53
Blakeley, J. A.	115, 125
von Bültzingsloewen, G.	189
Chakravarthy, S.	153, 165
Chang, P.	143
Dayal, U.	77
Deshpande, A.	125
Freytag, J. C.	41
Graefe, G.	1
Gunadhi, H.	159
Hammond, B.	181
Hasan, W.	85
Heiler, S.	65
Henschen, L. J.	59
Ioannidis, Y. E.	131
Kang, Y.	131
Kirkpatrick, J. E.	225
Krishnamurthy, R.	29, 209
van Kuijk, H. J. A.	53
Lafortune, S.	149
Lien, Y.	169
Lohman, G. M.	13
Luk, W. S.	197
Lyngbaek, P.	85
Martin, T. P.	203
Moss, J. E. B.	109
Muralikrishna, M.	175
Osborn, S. L.	91
Pathak, G.	115
Pirahesh, H.	71
Ramakrishnan, R.	37
Raschid, L.	235, 237
Rosenthal, A.	23
Roth, M. A.	225
Roussopoulos, N.	231
Scholl, M. H.	119
Segev, A.	159
Selinger, P. G.	19
Shaw, G. M.	95
Siegel, M. D.	47
Swami, A.	139
Tong, L.	231
Vandenberg, S.	101
Wang, X.	197
Weddell, G. E.	217
Wei, S.	169
Whang, K.	209
Wilkinson, K.	85
Yoo, H.	149
Yoon, S.	59
Zdonik, S. B.	95

Research Problems in Database Query Optimization

Goetz Graefe

Oregon Graduate Center
Beaverton, Oregon 97006-1999
graefe@cse.ogc.edu

In this overview, I briefly outline the areas in which I perceive the greatest need for query optimization research. I have structured this overview similarly to the workshop's discussion sections, starting with rule-based optimization and continuing with search techniques, selectivity estimation and cost models, and execution techniques. This paper is a survey of database query optimization; such a survey was written by Jarke and Koch [1]. Rather, this overview emphasizes open research questions and provides pointers to existing work.

1. Rule-Based Query Optimization

To prepare for future database applications, a number of research groups are working on extensible database systems [2, 3, 4, 5, 6]. For extensibility in these systems' query optimizers, a number of rule systems and languages have been designed and implemented [7, 8, 9, 10, 11]. The rule-languages in these approaches differ significantly, in particular in their level of abstraction and generality. For instance, while Lohman's rule system focuses on select-project-join queries [8], I attempted a more general approach that would work for arbitrary algebras of sets [9].

Not only do the rule languages and their underlying concepts differ, but also the modes of operation. Lohman et al. are working on a rule interpreter [12], while I found in a preliminary study that a C-Prolog interpreter as execution engine for a rule-based query optimizer was unacceptably slow [10]. This debate is clearly not closed; a final answer, I suspect, cannot be given without careful examination of the rule set, the operational environment and constraints, and the size of the search space. I contend, however, that implementing a special-purpose rule compiler is not harder or more cumbersome or less extensible than a special-purpose interpreter.

Any new query optimization system can only be realistically evaluated using experimental validation. While it is very difficult to develop a fair methodology for experimental comparison of relational database systems or their execution engines [13, 14, 15, 16], it will be at least as difficult to compare conventional or extensible query optimizers or extensible database systems in general. I hope to see some discussion, maybe in form of a conference panel, on evaluating the performance of extensible database systems.

Of course, we need not only rule languages and execution engines, we also need rules! For the strictly relational world, we seem to have a fairly good understanding of appropriate rules [17, 18]. Exceptions to this statement are probably necessary for nested SQL queries which are notoriously complex to optimize [19, 20, 21, 22, 23, 24] and disjunctive queries [25, 26]. Clearly, much more can be done in these areas. For nested relational systems and non-relational database systems, e.g., object-oriented systems, only limited work has been done so far [27, 28, 29, 30, 31]. Furthermore, for optimizing recursive queries, no rule sets have been published to-date [32, 33], although the Starburst group is working on such rules. A number of research groups, e.g. [34, 35, 36, 37], are pursuing query processing in deductive database.

We probably need multiple rule sets for the various stages of query compilation. First, we may wish to perform some optimization or normalization on the calculus level [1], assuming that queries are entered into the system in some calculus language. Second, we need to translate the calculus expression into an algebra expression [38]. Third, the algebra expression must be optimized and mapped to query

processing algorithms [8, 9]. Finally, the query evaluation plan must be interpreted or translated into an iterative program, e.g., using the techniques developed by Lorie [39] or Freytag [40, 41, 42, 43].

2. Search Techniques

Among the early, hard decisions to make when designing a query optimizer are what information to retrieve from the catalogs and derive for all intermediate results, e.g., cardinality and sort order, and the representation of alternative plans. One representation technique using dynamic programming was used in System R [44] and R* [45]. Unfortunately, it is not entirely clear to me how to use the technique when optimizing *bushy* trees, i.e., trees with composite inner relations as opposed to stored inner relations as required in System R, R*, or GAMMA [46].

In the search engine developed with the EXODUS optimizer generator, all plans are represented as trees with shared subtrees. While this representation is fairly compact, things can easily get out of hand when a transformation at a lower level in the tree requires reanalyzing multiple levels above [10]. The representation is of great importance since it determines how fast transformations can be performed and how fast duplicate derivation of the same alternative plan (using different rules or the same rules in a different sequence) can be detected.

Similarly, the detection of common subexpressions, in particular in multi-query or global query optimization, either on the calculus level of a query or in the algebra is facilitated by a suitable representation of plans [47, 48, 49, 50].

The next two problems concern the actual search. I assume that even though computer hardware gets faster, it will be impossible to perform an exhaustive search because the queries will become more complex and the search space larger, and database users expect lower response times from a database system running on faster hardware, not a longer search. The first problem concerns the order in which possible alternative plans explored, and the second one concerns pruning of alternative plans, i.e., decisions on which plans not to explore at all.

The order of transformations and exploration of alternatives is very important because the best plan found so far for a query or a subquery establishes an upper bound that can be used as a basis for pruning. Essentially this technique was used in the dynamic programming model of Selinger et al. [44], but it also applies to rule-based systems in which a bound may be included in a heuristic rather than used as an absolute bound.

Optimization heuristics are a wide-open field – it is well-known that selections should be done early and joins late. However, there are exceptions even to this simple rule when indices are involved. To my knowledge, there are no well-known rules proven in practice for the large variety of other relational algebra operations and query processing operations that are used in real database systems. For example, when should one do projections, aggregation, duplicate elimination, union, or intersection? And to make the problem "just a little bit" harder, are there suitable heuristics for transitive closure and other recursion operations?

One approach to the large search space of complex queries is to use multiple phases in the optimization [51]. For large queries, it turns out that multiple phases can find better plans faster. For example, after modifying the EXODUS optimizer generator's search engine, some improvements in search speed and plan quality could be achieved by using two passes with different rule sets [10].

Now that I have outlined the difficulties, is there some form of relief in parallel search? Are there suitable query plan representations and search strategies that allow parallel search on multi-processors? For the time being, I would be thrilled to see effective search algorithms for shared-memory machines.

Finally, when all is done, not all is yet said. Besides an investigation of optimizer performance [52], I know of only one effort to validate a query optimizer, namely Mackert and Lohman's work [53, 54]. There was some very preliminary work on a relational optimizer benchmark within the GAMMA project by DeWitt, Muralikrishna, and myself, but we never finished this sub-project. While Mackert and Lohman carefully inspected the accuracy of R*'s cost model, they did not validate other important characteristics of their optimizer, e.g., whether the set of algorithms in R* really represented an optimal selection as suggested by Blasgen and Eswaran [55], or whether the search space was sufficiently large.

In fact, a remark by Selinger et al. [44] and my comparison of bushy vs. left-deep trees [10] suggest that the optimizer and the run-time performance could have improved by including composite inner relations. The Starburst optimizer will have a run-time switch whether or not to consider composite inner relations in joins [56].

3. Selectivity Estimation and Cost Functions

All realistic query optimization hinges on accurate cost estimation. Cost estimation requires estimates of relation, or, more generally, set sizes. Wong and Youssefi [57, 58] assumed that accurate selectivity estimation is so hard that it cannot be done before query evaluation, and developed a query processing strategy that intertwined planning and execution.

A large number of researchers have worked on selectivity estimation [45, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71]. Other work focused on skewed distributions [72], on correlated attributes [73, 74], and on block selectivities [75, 76]. Despite the significant effort spent, a large number of problems remains unsolved. Estimation techniques are only estimations; therefore, their accuracy and reliability is open to question. At the current time, I am aware of three approaches to the reliability problem. First, one can argue that a limited estimation error does not make much practical difference [77]. Second, one can intertwine optimization and execution, either by deciding on one step at a time [57] or by executing a number of operations, e.g., five joins, and then reconsidering what steps to perform next [78]. Third, one can design an adaptive or dynamic scheme to handle estimation errors [79]. Dynamic schemes also allow effective evaluation of embedded queries with run-time variables in the query predicate or varying system loads [80].

Even a very reliable estimation scheme will not suffice for very large queries with many (e.g., more than a dozen) cascaded operations, since the errors typically multiply. Very reliable estimation schemes or dynamic plans have to be combined to determine the optimal or a near-optimal plan for such queries [81, 82]. Possible relief for the problem may come from using bushy trees which are not as deep as left-deep trees and therefore do not have as many factors in the error multiplication.

Reliable estimation of intermediate result sizes is also necessary to determine a priori what resources should be committed to an operation. Several buffer management algorithms require that working set sizes be declared a priori [83, 84, 85], and knowledge of intermediate set sizes can help in determining optimal degrees of parallelism in parallel dataflow query evaluators [86].

Hardly any work has been done in estimating the size of transitive closures and other recursive operations. While a fair number of execution algorithms and their complexities are known [87, 88], query optimization technology lags behind because we do not know how to estimate the input variables to the complexity formulas. To the best of my knowledge, only Naughton has worked on the subject. My own, rather limited experiments with two-dimensional histograms did not lead to conclusive results.

Another stone that has been left virtually unturned is selectivity estimation for nested relations and sets of complex objects. Some of the join selectivity work and some of the transitive closure selectivity work (when it is completed!) will assist in these areas, but they probably will not suffice for efficient, effective, and reliable estimation schemes.

Finally, in order to build a complete extensible database system, not only an extensible (rule-based) optimizer but also an extensible selectivity estimation scheme is needed. Work in this area has been very limited so far [66], and more needs to be done, including a practical evaluation.

4. Execution Techniques

Researchers and implementors of query optimizers must be keenly aware of query evaluation techniques. After all, the objective of query optimization is to choose from a set of available processing methods.

Traditionally, query optimization has focused on conjunctive or select-project-join queries. The set of join methods to be considered remained basically unchallenged and restricted to nested loops and merge join, with or without indices, locally or on parallel machines, implemented in software or in hardware [89, 90, 91, 92, 93, 94, 95]. Only recently have hash-based methods gained popularity, mainly

within research efforts [96, 97, 98]. Interestingly, hash joins are frequently associated with multiprocessor query evaluation systems [99, 100, 101]. Of course, the basic hash join algorithm is nothing more than a index-nested-loops join with a very efficient index. Only the variants of hash join make it interesting to query optimizer implementors.

The main problem with hash joins is the hash table: will it fit in main memory? There are two methods for dealing with hash table overflow (or any other problem, for that matter), avoidance and resolution. Both methods can be designed around partitioning, either before the actual hash table is built [101] or while it is built [98, 102]. If the query optimizer's selectivity estimator is sufficiently accurate and reliable, more efficient overflow avoidance techniques than proposed to-date should be possible.

One way to avoid hash table overflow is to partition the join input over more memory by employing multiple machines (which might also reduce response times), as was done in the GAMMA project [46]. Designing a query optimizer that decides on appropriate degrees of vertical parallelism (pipelining) and horizontal parallelism (partitioning) has not been addressed sufficiently. A thorough comparison of sort- and hash-based join algorithms was provided by Shapiro [103].

Aggregate functions (e.g., sum of salaries by department) are equality-based; therefore, they can be implemented using hashing. Duplicate removal is just a special-form aggregate function, and duplicate removal algorithms can be modified to perform aggregate functions and vice versa. Only limited work hash been spent on aggregate functions, both in query optimization and execution [104, 105, 106, 20, 23, 107, 108, 109]. A single hash function is faster to evaluate than multiple ($\log N$) comparisons, and unless the sort operation is implemented quite cleverly, it requires more I/O than hashing. Avoiding hash table overflow requires that the aggregate output fits in main memory; aggregating in memory by sorting requires that the input fit in memory, even if duplicates are aggregated while writing runs.

For all parallel algorithms, partitioning is of great importance. Query optimizers should be able to choose from a variety of partitioning methods and parameters to ensure balanced load for all processors [110, 111].

Recursive joins have been studied for some time now, both on a single processors [87, 112, 113, 114, 115] and on multi-processors [116, 117]. We probably will have to wait until such algorithms are included in complete database systems before we can evaluate them realistically. Benchmarks for transitive closure algorithms are urgently needed.

There are many other algorithms on sets that should be included in database systems, including semi-join, outer-join, union, intersection, difference [118], division [119], and algorithms for nested relations. We are currently building a research vehicle to experiment with these operations both on single- and multi-processor systems [120, 121].

Another concern is physical database organization, including indexing [122], clustering [123, 124, 125, 126, 127, 128], declustering [129, 130], and shadowing [131]. Clustering techniques for complex objects have drawn special attention [132, 133].

Finally, distributed and heterogeneous database systems pose many additional open problems [45, 134, 21, 135, 136, 137, 138]. I suspect that for heterogeneous and federated systems, extensible rule-based optimizers will prove very useful. The design of independent, non-interfering rule sets that can be combined as database systems are linked together is a new, very hard challenge. Maier suggested that autonomous systems could use a rule language to communicate about their query processing capabilities [139].

Summary

There is a large number of problems that we have to work on. I have tried to give a very brief overview of a fair number of them, and at the same time point to existing work – I am sure that I have omitted a great number of researchers and short-changed some others. I must ask for their apologies, and hope that they will point me in directions that I have failed to explore so far.

References

1. M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys* **16**(2) pp. 111-152 (June 1984).
2. M.J. Carey, D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.T. Schuh, E.J. Shekita, and S. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview," in *Readings on Object-Oriented Database Systems*, ed. D. Maier and S. Zdonik,Morgan Kaufman, San Mateo, CA. (1989).
3. P. Schwarz, W. Chang, J.C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, "Extensibility in the Starburst Database System," *Proceedings of the Int'l Workshop on Object-Oriented Database Systems*, pp. 85-92 (September 1986).
4. M. Stonebraker and L.A. Rowe, "The Design of POSTGRES," *Proceedings of the ACM SIGMOD Conference*, pp. 340-355 (May 1986).
5. H.B. Paul, H.J. Schek, M.H. Scholl, G. Weikum, and U. Deppisch, "Architecture and Implementation of the Darmstadt Database Kernel System," *Proceedings of the ACM SIGMOD Conference*, pp. 196-207 (May 1987).
6. D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise, "GENESIS: An Extensible Database Management System," *IEEE Transactions on Software Engineering* **SE-14**(11) p. 1711 (November 1988).
7. J.C. Freytag, "A Rule-Based View of Query Optimization," *Proceedings of the ACM SIGMOD Conference*, pp. 172-180 (May 1987).
8. G.M. Lohman, "Grammar-Like Functional Rules for Representing Query Optimization Alternatives," *Proceedings of the ACM SIGMOD Conference*, pp. 18-27 (June 1988).
9. G. Graefe and D.J. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the ACM SIGMOD Conference*, pp. 160-171 (May 1987).
10. G. Graefe, "Rule-Based Query Optimization in Extensible Database Systems," *Ph.D. Thesis*, University of Wisconsin, (August 1987).
11. G. Graefe, "Software Modularization with the EXODUS Optimizer Generator," *IEEE Database Engineering*, (December 1987).
12. M. Lee, J. Freytag, and G. Lohman, "Implementing an Interpreter for Functional Rules in a Query Optimizer," *Proceedings of the Conference on Very Large Databases*, pp. 218-229 (August 1988).
13. D. Bitton, D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach," *Proceeding of the Conference on Very Large Data Bases*, pp. 8-19 (October-November 1983).
14. H. Boral and D.J. DeWitt, "A Methodology for Database System Performance Evaluation," *Proceedings of the ACM SIGMOD Conference*, pp. 176-185 (June 1984).
15. Anon. et al., "A Measure of Transaction Processing Power," *Datamation*, pp. 112-118 (April 1, 1985).
16. C. Turbyfill, *Comparative benchmarking of relational database systems*, Ph.D. thesis, Cornell University (January 1988).
17. J.D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, MD. (1982).
18. D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, MD. (1983).
19. W. Kim, "On Optimizing an SQL-like Nested Query," *ACM Transactions on Database Systems* **7**(3) pp. 443-469 (September 1982).
20. W. Kiessling, "On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates," *Proceedings of the Conference on Very Large Data Bases*, pp. 241-250 (August 1985).
21. G.M. Lohman, D. Daniels, L.M. Haas, R. Kistler, and P.G. Selinger, "Optimisation of Nested Queries in a Distributed Relational Database," *Proceedings of the Conference on Very Large Data*

Bases, pp. 403-415 (August 1984).

22. R.A. Ganski and H.K.T. Wong, "Optimization of Nested SQL Queries Revisited," *Proceedings of the ACM SIGMOD Conference*, pp. 23-33 (May 1987).

23. U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries that contain Nested Subqueries, Aggregates, and Quantifiers," *Proceeding of the Conference on Very Large Data Bases*, pp. 197-208 (August 1987).

24. M.M. Krishna, "Optimizing and Dataflow Algorithms for Nested Tree Queries," *Proceedings of the Conference on Very Large Databases*, (August 1989).

25. M. Muralikrishna and D. DeWitt, "Optimization of Multiple-Relation Multiple-Disjunct Queries," *Proceedings of the 7th SIGACT-SIGMOD Symposion on Principles of Database Systems*, pp. 263-275 (March 1988).

26. F. Bry, "Toward an Efficient Evaluation of General Queries: Quantifers and Disjunction Processing Revisited," *Proceedings of the ACM SIGMOD Conference*, (May-June 1989).

27. S.L. Osborn, "Identity, Equality, and Query Optimization," pp. 346-351 in *Advances in Object-Oriented Database Systems*, ed. K.R. Dittrich,Springer-Verlag (September 1988).

28. S. Zdonik, "Data Abstraction and Query Optimization," pp. 368-373 in *Advances in Object-Oriented Database Systems*, ed. K.R. Dittrich,Springer-Verlag (September 1988).

29. H.F. Korth, "Optimization of Object-Retrieval Queries," pp. 352-357 in *Advances in Object-Oriented Database Systems*, ed. K.R. Dittrich,Springer-Verlag (September 1988).

30. G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: A Prospectus," pp. 358-363 in *Advances in Object-Oriented Database Systems*, ed. K.R. Dittrich,Springer-Verlag (September 1988).

31. L. Colby, "A Recursive Algebra and Query Optimization for Nested Relations," *Proceedings of the ACM SIGMOD Conference*, (May-June 1989).

32. Y.E. Ioannidis, "Commutativity and its role in the processing of linear recursion," Technical Report #804, University of Wisconsin-Madison (November 1988).

33. Y.E. Ioannidis and E. Wong, "Towards an algebraic theory of recursion," Technical Report #801, University of Wisconsin-Madison (October 1988).

34. S. Tsur and C. Zaniolo, "LDL: A Logic-Based Data-Language," *Proceeding of the Conference on Very Large Data Bases*, pp. 33-41 (August 1986).

35. K. Morris, "An Algorithm for Ordering Subgoals in NAIL!," *Proceedings of the 7th SIGACT-SIGMOD Symposion on Principles of Database Systems*, pp. 82-88 (March 1988).

36. U.S. Chakravarthy, J. Grant, and J. Minker, "Foundations of Semantic Query Optimization for Deductive Databases," pp. 243-273 in *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker,Morgan-Kaufman, Los Algos, CA. (1988).

37. F. Bancilhon and R. Ramakrishnan, "Performance Evaluation of Data Intensive Logic Programs," pp. 439-517 in *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker,Morgan-Kaufman, Los Algos, CA. (1988).

38. S. Ceri and G. Gottlob, "Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries," *IEEE Transactions on Software Engineering* **SE-11**(4) pp. 324-345 (April 1985).

39. R.A. Lorie and J.F. Nilsson, "An Access Specification Language for a Relational Database Management System," *IBM Journal of Research and Development* **23**(3) pp. 286-298 (May 1979).

40. J.C. Freytag, "Translating Relational Queries into Iterative Programs," *Ph.D. Thesis*, Harvard University, (September 1985).

41. J.C. Freytag and N. Goodman, "Rule-Based Transformation of Relational Queries into Iterative Programs," *Proceedings of the ACM SIGMOD Conference*, pp. 206-214 (May 1986).

42. J.C. Freytag and N. Goodman, "Translating Aggregate Queries into Iterative Programs," *Proceeding of the Conference on Very Large Data Bases*, pp. 138-146 (August 1986).

43. J.C. Freytag and N. Goodman, "On the Translation of Relational Queries into Iterative Programs," *ACM Transaction on Database Systems* **14**(1) pp. 1-27 (March 1989).

44. P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD Conference*, pp. 23-34 (May-June 1979).

45. P. Griffiths Selinger and M. Adiba, "Access Path Selection in Distributed Database Management Systems," *Computer Science Research Report*, (RJ2883)IBM Research Laboratory, (August 1980).

46. D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," *Proceedings of the Conference on Very Large Data Bases*, pp. 228-237 (August 1986).

47. M. Jarke, "Common Subexpression Isolation in Multiple Query Optimization," pp. 191-205 in *Query Processing in Database Systems*, ed. W. Kim, D.S. Reiner, and D.S. Batory,Springer, Berlin (1985).

48. J. Park and A. Segev, "Using Common Subexpressions To Optimize Multiple Queries," *Proceedings of the IEEE Conference on Data Engineering*, pp. 311-319 (February 1988).

49. T.K. Sellis, "Multiple-Query Optimization," *ACM Transaction on Database Systems* **13**(1) pp. 23-52 (March 1988).

50. A. Rosenthal and U. Chakravarthy, "Anatomy of a Modular Multiple Query Optimizer," *Proceedings of the Conference on Very Large Databases*, pp. 230-239 (August 1988).

51. A. Rosenthal, U. Dayal, and D. Reiner, "Fast Query Optimization Over a Large Strategy Space: The Pilot Pass Approach," *unpublished manuscript*, (1986).

52. M.M. Astrahan, W. Kim, and M. Schkolnik, "Performance of the System R Access Path Selection Mechanism," *Proceedings IFIP Congress*, pp. 487-491 (1980).

53. L.F. Mackert and G.M. Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries," *Proceedings of the ACM SIGMOD Conference*, pp. 84-95 (May 1986).

54. L.F. Mackert and G.M. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," *Proceedings of the Conference on Very Large Data Bases*, pp. 149-159 (August 1986).

55. M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases," *IBM Systems Journal* **16**(4)(1977).

56. K. Ono and G.M. Lohman, "Extensible Enumeration of Feasible Joins for Relational Query Optimization," *IBM Research Report* **RJ 6625 (63936)**(December 1988).

57. E. Wong and K. Youssefi, "Decomposition - A Strategy for Query Processing," *ACM Transactions on Database Systems* **1**(3) pp. 223-241 (September 1976).

58. K. Youssefi and E. Wong, "Query Processing in a Relational Database Management System," *Proceedings of the Conference on Very Large Data Bases*, pp. 409-417 (October 1979).

59. S. Christodoulakis, "Estimating Selectivities in Databases," *Ph.D. Thesis*, University of Toronto, (1981).

60. P. Richard, "Evaluation of the Size of a Query Expressed in Relational Algebra," *Proceedings of the ACM SIGMOD Conference*, pp. 155-163 (April-May 1981).

61. R. Demolombe, "Estimation of the Number of Tuples Satisfying a Query Expressed in Predicate Calculus Language," *Proceedings of the Conference on Very Large Data Bases*, pp. 55-63 (October

1980).

62. S. Christodoulakis, "Estimating Record Selectivities," *Information Systems* **8**(2) pp. 105-115 (1983).

63. N.C. Rowe, "Top-Down Statistical Estimation on a Database," *Proceedings of the ACM SIGMOD Conference*, pp. 135-145 (May 1983).

64. G. Piatetsky-Shapiro and C. Connell, "Accurate Estimation of the Number of Tuples Satisfying a Condition," *Proceedings of the ACM SIGMOD Conference*, pp. 256-276 (June 1984).

65. D. Yang, "Expections Associated with Compound Selection and Join Operations," *Ph.D. Thesis, Computer Science Technical Report*, (RM-85-02)University of Virginia, (July 1985).

66. M.V. Mannino and A. Rivera, "An extensible model of selectivity estimation," *to appear Information Sciences, Spring 1988*, Center for Business Decision Analysis, (December 1987).

67. M.M. Astrahan, M. Schkolnick, and K.Y. Whang, "Approximating the number of unique values of an attribute without sorting," *Information Systems* **12**(1) p. 11 (1987).

68. G. Graefe, "Selectivity Estimation Using Moments and Density Functions," *Oregon Graduate Center, Computer Science Technical Report*, (87-012)(November 1987).

69. W-C. Hou, G. Ozsoyoglu, and B. Taneja, "Statistical Estimators for Relational Algebra Expressions," *Proceedings of the 7th SIGACT-SIGMOD Symposion on Principles of Database Systems*, pp. 276-287 (March 1988).

70. M.V. Mannino, P. Chu, and T. Sager, "Statistical Profile Estimation in Database Systems," *ACM Computing Surveys* **20**(3)(September 1988).

71. R. Ahad, K.V. Bapa Rao, and D. McLead, "On Estimating the Cardinality of the Projection of a Database Relation," *ACM Transaction on Database Systems* **14**(1) pp. 28-40 (March 1989).

72. C. Lynch, "Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values," *Proceedings of the Conference on Very Large Databases*, pp. 240-251 (August 1988).

73. B.T. Vander Zanden, H.M. Taylor, and D. Bitton, "Estimating Block Accesses When Attributes Are Correlated," *Proceeding of the Conference on Very Large Data Bases*, pp. 119-127 (August 1986).

74. D.A. Bell, D.H.O. Ling, and S. McClean, "Pragmatic Estimation of Join Sizes and Attribute Correlations," *Proceedings of the IEEE Conference on Data Engineering*, p. 76 (February 1989).

75. S. Christodoulakis, "Estimating Block Selectivities," *Information Systems* **9**(1) p. 69 (1984).

76. B.T. Vander Zanden, H.M. Taylor, and D. Bitton, "A general framework for computing block accesses," *Information Systems* **12**(2) p. 177 (1987).

77. A. Kumar and M. Stonebraker, "The Effect of Join Selectivities on Optimal Nesting Order," *SIGMOD Record* **16**(1) pp. 28-41 (March 1987).

78. D.J. DeWitt, *Personal Communication.* March 1988.

79. Y.H. Lee and P.S. Yu, "Adaptive Selection of Access Path and Join Method," *unpublished manuscript*, (1988).

80. G. Graefe and K. Ward, "Dynamic Query Evaluation Plans," *Proceedings of the ACM SIGMOD Conference*, (May-June 1989).

81. A. Swami and A. Gupta, "Optimizing Large Join Queries," *Proceedings of the ACM SIGMOD Conference*, pp. 8-17 (June 1988).

82. A. Swami, "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques," *Proceedings of the ACM SIGMOD Conference*, (May-June 1989).

83. G.M. Sacco and M. Schkolnik, "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model," *Proceeding of the Conference on Very Large Data Bases*, pp. 257-262 (September 1982).

84. H.T. Chou, "Buffer Management of Database Systems," *Ph.D. Thesis*, University of Wisconsin, (May 1985).

85. H.T. Chou and D.J. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proceedings of the Conference on Very Large Data Bases*, pp. 127-141 (August 1985).

86. M. Stonebraker, P. Aoki, and M. Seltzer, "Parallelism in XPRS," *UCB/ERL Memorandum M89/16*, University of California, (February 1989).

87. F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," *Proceedings of the ACM SIGMOD Conference*, pp. 16-52 (May 1986).

88. R. Agrawal, "Alpha: An Extension of Relational Algebra To Express a Class of Recursive Queries," *Proceedings of the IEEE Conference on Data Engineering*, p. 580 (February 1989).

89. P. Valduriez, "Semi-Join Algorithms for Multiprocessor Systems," *Proceedings of the ACM SIGMOD Conference*, pp. 225-233 (June 1982).

90. H. Lu and M. Carey, "Some Experimental Results on Distributed Join Algorithms in a Local Network," *Proceedings of the Conference on Very Large Data Bases*, pp. 292-304 (August 1985).

91. P. Valduriez, "Join Indices," *ACM Transaction on Database Systems* **12**(2) pp. 218-246 (June 1987).

92. C.K. Baru, O. Frieder, D. Kandlur, and M. Segal, "Join on a Cube: Analysis, Simulation, and Implementation," *Proceedings of the 5th International Workshop on Database Machines*, (1987).

93. E.R. Omiecinski, "Heuristics for Join Processing Using Nonclustered Indices," *IEEE Transactions on Software Engineering* **SE-15**(1) p. 18 (January 1989).

94. M. Kitsuregawa, L. Harada, and M. Takagi, "Join Strategies on KD-Tree Indexed Relations," *Proceedings of the IEEE Conference on Data Engineering*, p. 85 (February 1989).

95. B.C. Desai, "Performance of a Composite Attribute and Join Index," *IEEE Transactions on Software Engineering* **SE-15**(2) p. 142 (February 1989).

96. K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations," *Proceedings of the Conference on Very Large Data Bases*, pp. 323-333 (August 1984).

97. M. Kitsuregawa et al., "Architecture and performance of relational algebra machine GRACE," *Proc. Int. Conf. Parallel Processing*, pp. 241-250 (1984).

98. D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD Conference*, pp. 1-8 (June 1984).

99. R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," *Ph.D. Thesis*, University of Wisconsin, (October 1986).

100. S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An Overview of The System Software of A Parallel Relational Database Machine GRACE," *Proceeding of the Conference on Very Large Data Bases*, pp. 209-219 (August 1986).

101. M. Nakayama, M. Kitsuregawa, and M. Takagi, "Hash-Partitioned Join Method Using Dynamic Destaging Strategy," *Proceedings of the Conference on Very Large Databases*, pp. 468-478 (August 1988).

102. D.J. DeWitt and R.H. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proceedings of the Conference on Very Large Data Bases*, pp. 151-164 (August 1985).

103. L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Transactions on Database Systems* **11**(3) pp. 239-264 (September 1986).

104. R. Epstein, "Techniques for Processing of Aggregates in Relational Database Systems," *UCB/ERL Memorandum*, (M79/8)University of California, (February 1979).

105. A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *Journal of the ACM* **29**(3) pp. 699-717 (July 1982).

106. D. Bitton and D.J. DeWitt, "Duplicate Record Elimination in Large Data Files," *ACM Transactions on Database Systems* **8**(2) pp. 255-265 (June 1983).

107. G. von Bultzingsloewen, "Translating and Optimizing SQL Queries Having Aggregates," *Proceeding of the Conference on Very Large Data Bases*, pp. 235-244 (August 1987).

108. J. Bradley, "A Group-Select Operation for Relational Algebra and Implications for Database Machine Design," *IEEE Transactions on Software Engineering* **SE-14**(1) p. 126 (January 1988).

109. J. Srivastava and V.Y. Lum, "A Tree Based Access Method (TBSAM) for Fast Processing of Aggregate Queries," *Proceedings of the IEEE Conference on Data Engineering*, pp. 504-510 (February 1988).

110. D. Sacca and G. Wiederhold, "Database Partitioning in a Cluster of Processors," *Proceeding of the Conference on Very Large Data Bases*, pp. 241-247 (October-November 1983).

111. D. Sacca and G. Wiederhold, "Database Partitioning in a Cluster of Processors," *ACM Transactions on Database Systems* **10**(1) pp. 29-56 (March 1985).

112. R. Agrawal and H. Jagadish, "Direct Algorithms for Computing the Transitive Closure of Database Relations," *Proceeding of the Conference on Very Large Data Bases*, pp. 255-266 (August 1987).

113. J.A. Thom, K. Ramamohanarao, and L. Naish, "A Superjoin Algorithm for Deductive Databases," pp. 519-543 in *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker,Morgan-Kaufman, Los Algos, CA. (1988).

114. Y. Ioannidis, "On the Computation of the Transitive Closure of Relational Operators," *Proceeding of the Conference on Very Large Data Bases*, pp. 403-411 (August 1986).

115. Y. Ioannidis and R. Ramakrishnan, "Efficient Transitive Closure Algorithms," *Proceedings of the Conference on Very Large Databases*, pp. 382-394 (August 1988).

116. D.A. Schneider and M.J. Skarpelos, "Design and Implementation of a Distributed Transitive Closure Algorithm," *Computer Sciences 764 Course Project*, University of Wisconsin, (May 1986).

117. R. Agrawal and H.V. Jagadish, "Multiprocessor Transitive Closure Algorithms," *IEEE Database Engineering* **12**(1) pp. 58-64 (March 1989).

118. T. Keller and G. Graefe, "The One-to-One Match Operator of the Volcano Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-009)(June 1989).

119. G. Graefe, "Relational Division: Four Algorithms and Their Performance," *Proceedings of the IEEE Conference on Data Engineering*, pp. 94-101 (February 1989).

120. G. Graefe, "Volcano: An Extensible and Parallel Dataflow Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-006)(June 1989).

121. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-007)(June 1989).

122. M. Hammer and A. Chan, "Index Selection in a Self-Adaptive Data Base Management System," *Proceedings of the ACM SIGMOD Conference*, pp. 1-8 (1976).

123. C.T. Yu, K. Lam, M.K. Siu, and M. Ozsoyoglu, "Performance Analysis of Three Related Assignment Problems," *Proceedings of the ACM SIGMOD Conference*, pp. 82-92 (May-June 1979).

124. J.M. Chang and K.S. Fu, "A Dynamic Clustering Technique for Physical Database Design," *Proceedings of the ACM SIGMOD Conference*, pp. 188-199 (May 1980).

125. C.T. Yu, C.M. Suen, K. Lam, and M.K. Siu, "Adaptive Record Clustering," *ACM Transactions on Database Systems* **10**(2) pp. 180-204 (June 1985).

126. S. Fushimi, M. Kitsuregawa, M. Nakayama, H. Tanaka, and T. Moto-oka, "Algorithm and Performance Evaluation of Adaptive Multidimensional Clustering Technique," *Proceedings of the ACM SIGMOD Conference*, pp. 308-318 (May 1985).

127. J.P. Cheiney and G. Kiernan, "A Functional Clustering Method for Optimal Access to Complex Domains in a Relational DBMS," *Proceedings of the IEEE Conference on Data Engineering*, pp. 394-401 (February 1988).

128. C.T. Yu and T.M. Jiang, "Adaptive Algorithms for Balanced Multidimensional Clustering," *Proceedings of the IEEE Conference on Data Engineering*, pp. 386-393 (February 1988).

129. M.T. Fang, R.C.T. Lee, and C.C. Chang, "The Idea of Declustering and Its Applications," *Proceeding of the Conference on Very Large Data Bases*, pp. 181-188 (August 1986).

130. K. Salem and H. Garcia-Molina, "Disk Striping," *EECS Techical Report 332*, Princeton University, (December 1984).

131. D. Bitton and J. Gray, "Disk Shadowing," *Proceedings of the Conference on Very Large Databases*, pp. 331-338 (August 1988).

132. J. Banerjee, W. Kim, S.J. Kim, and J.F. Garza, "Clustering a DAG for CAD Databases," *IEEE Transactions on Software Engineering* **SE-14**(11) p. 1684 (November 1988).

133. E. Chang and R. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS," *Proceedings of the ACM SIGMOD Conference*, (May-June 1989).

134. C.W. Chung and K. Irani, "A Methodology for Query Optimization in Distributed Database Systems," *IEEE Database Engineering* **5**(3)(September 1982).

135. A.L.P. Chen and V.O.K. Li, "Optimising Star Queries in a Distributed Database System," *Proceedings of the Conference on Very Large Data Bases*, pp. 429-437 (August 1984).

136. B. Gavish and A. Segev, "Set Query Optimization in Distributed Database Systems," *ACM Transactions on Database Systems* **11**(3) pp. 265-293 (September 1986).

137. P. Bodorik and J.S. Riordon, "Distributed Query Processing Optimization Objectives," *Proceedings of the IEEE Conference on Data Engineering*, pp. 320-329 (February 1988).

138. S. Pramanik and D. Vineyard, "Optimizing Join Queries in Distributed Databases," *IEEE Transactions on Software Engineering* **SE-14**(9) p. 1319 (September 1988).

139. D. Maier, *Personal Communication*. May 1989.

Is Query Optimization a "Solved" Problem?

Guy M. Lohman
IBM Almaden Research Center
San Jose, CA 95120
lohman@ibm.com

Where *is* everybody?

Despite a growing reliance on query optimization technology pioneered only a decade ago [WONG 76], [SELI 79], there seems to be a dearth of innovative yet soundly validated research in query optimization. Now that introductory database textbooks have an entire chapter on the subject, has query optimization matured to the point that it is an engineering, rather than a research, topic? Or are the remaining problems just so hard (maybe even insoluble?) that no one dares attack them? Could industry be wooing would-be researchers away from academia and locking them up in a cloak of trade secrecy to protect any advantage that advances in this technology might give their products?

I'm baffled by the relatively small community of researchers working in query optimization, given the wealth of topics that I see remaining. Rather than diminishing, I see the challenge increasing, particularly with the current emphasis of many groups on "extensibility". It seems to me that my colleagues can dream up new and wonderful extensions much faster than I can come up with ways to deal with all the optimization issues that those extensions always raise. This extended abstract attempts to list and classify what I consider are some of the more interesting of these issues, in order to attract more attention to this important area. At the risk of goring someone's favorite ox, I will also enumerate a few areas in which I see very little future potential.

Cardinality Estimation

Estimating the cardinality of intermediate and final results of a query remains the "Achilles heel" of query optimization. The cost of any operation is invariably proportional to how often that operation must be performed, i.e. how many tuples it processes. Worse, errors in cardinality estimates tend to compound for larger queries: the estimate for ((A ⋈ B) ⋈ C) ⋈ D is proportional to the estimate for (A ⋈ B) ⋈ C, which in turn is proportional to the estimate for A ⋈ B. The probabilistic models used by virtually every query optimizer are flawed by two major assumptions: (1) uniform distribution of values and (2) independence. The first assumption has been attacked, at least for single-table predicates, by keeping more detailed statistics such as histograms [PIAT 84], higher order moments [GRAE 87c], and the frequency of the N most common values [LYNC 88]. With a few exceptions ([CHRI 83], [VAND 86], [MURA 88]), the assumption of attribute independence has rarely been relaxed. For example, the predicates SALARY > $100K and AGE > 40 are dependent in subtle ways determined by the semantics of the underlying database; multiplying the individual selectivities for each predicate will underestimate the number of number of tuples actually in the database. When compiled queries are going to be run often enough to justify the additional optimization time, it may be worthwhile to visit the database itself, either using auxiliary structures such as indices (as IBM's SQL/400 product does [IBM]!) or sampling [PIAT 84].

Few of these schemes address the harder problem of estimating the selectivity of joins. If we are willing to endure the costs of maintenance and the resulting contention of updates, join indices [VALD 87] could yield (almost) exact cardinalities, in the same way described above for single-table indices and single-table predicates. Selinger et al. estimated the join selectivity as the inverse of the maximum of the two join-column cardinalities (number of distinct values), essentially assuming that one set is a subset of the other, i.e. that each key value in the smaller set has a matching value in the larger set [SELI 79]. While this assumption is correct for key domains having referential integrity constraints, in general the overlap of the two sets may vary greatly depending upon the semantics of the domain.

Even if we could solve the above problems, predicates involving host variables, whose values are determined at run-time by the application program, remain a vexing problem for compile-time optimization. Graefe has proposed keeping alternative execution plans, the choice of which depends upon the value supplied by the program [GRAE 89], but the number of different values that could be provided might require a similarly large number of alternative plans. The host-variable dilemma is only the worst subset of the much broader difficulty for any compile-time optimizer to know in advance the run-time milieu of its queries, e.g. the characteristics of other queries competing with this one and the resources (e.g., main memory) available.

Beyond these age-old problems that have yet to see a comprehensive solution, extensions add a legion of new woes. How do user-defined functions on user-defined domains convey selectivities to the optimizer? Should *any* new domain have to include this information, or could it be inherited from the containing domain, subject to some rules? For example, selectivities on the "kilometers" domain should be inheritable from the "positive reals" domain, but selectivities on the "date" domain cannot readily be inherited from the "natural numbers" domain: special rules apply! Recursive queries add another unknown dimension: estimating the number of iterations to process the recursion, as well as how many new tuples are likely to be generated at each iteration. We are investigating all of these problems in the Starburst project, but don't have many good solutions at this point.

Rule-based Query Optimization

Few people question the flexibility gained by using rules to specify alternative execution plans, and there has been a lot of pioneering work within the last few years [GRAE 87a], [FREY 87], [GRAE 87b]. Debate continues, however, on what form those rules should take. In Starburst, two different kinds of rules are used at two different levels of query processing:

1. At a fairly macro level (e.g., treating a simple SELECT-PROJECT-JOIN as a single operator), context-dependent rules written in C transform the semantic representation of any query into a semantically equivalent but "better" query [HASA 88]. These transformations include merging views, converting subqueries to joins (when possible), and some predicate pushdown[1]. Since *methods* for performing operations are not determined by this level, no cost model is involved. When a transformation is not clearly beneficial, both the original and transformed representations are retained and connected by a CHOOSE operator [GRAE 89], [HASA 88]; the Plan Generator will evaluate both and choose the cheaper. The "context" on which these transformations depend are the neighboring operations to which a given operation relates, e.g. which use its results or provide a correlation value.

---

1 Applying a join predicate via an index on an individual table depends upon which table is the outer table of that join, so this form of "predicate pushdown" is determined by the second set of rules.

2. Within each macro operation, the Plan Generator interprets a grammar-like set of production rules (called STrategy Alternative Rules, or STARs) to construct and analyze the properties of alternative Query Execution Plans (QEPs), which are trees of executable operators (called LOw-LEvel Plan OPerators, or LOLEPOPs) [LOHM 88]. The STARs are simplified, and hence can be interpreted quickly [LEE 88], because they are independent of the context in which they are applied. The properties of any QEP includes its estimated cost.

Having different kinds of rules for different kinds of optimization allows us to customize the rules to exploit the structure of each kind of optimization, but raises several interface issues that we haven't resolved yet. Will the alternatives remaining after semantic optimization proliferate unmanageably as more rules are added? Will we need to "ping-pong" between semantic and plan optimization? How well do both optimizations really perform? Ideally, we would like to find some unified set of rules that could be input as data to the optimizer, yet could be interpreted efficiently by a single rule processor. Could this more general form of rules even support knowledge base applications and a more "active" database [DAYA 88]? Or does "one size fits all" fit none well?

Currently, Starburst's semantic optimizer is completely implemented and we are working on extensions. The rule interpreter of the Plan Generator is operational and producing executable plans for a simplified set of single-table rules, and we are implementing some of the infrastructure needed for join rules to store and retrieve the best plan fragments.

Search Strategies

There has been much hand-wringing but little empirical testing of the worst-case combinatorics of System R's dynamic programming algorithm for optimization. Recent experiments on Starburst's Join Enumerator, which also uses dynamic programming, proves that the complexity of optimization changes dramatically depending upon the "shape" of the query graph. In the quite common case of a "linear" query graph, the complexity is polynomial in the number of tables, whereas "star" query graphs are effectively equivalent to complete graphs for purposes of optimization [ONO 88].

Complexity is also affected by whether the inner table of a join can be composite, i.e. the result of a join, something controlled by an application-specified parameter in Starburst. Other such parameters give applications considerable control over the search mechanism by allowing or deferring Cartesian products, pruning alternatives from STARs, and even controlling the order in which STARs are evaluated. Interactive queries will want to minimize the size of the search space by eliminating composite inners, deferring Cartesian products, and pruning all but the most probable alternatives, whereas compiled queries that will be run quite often can afford the luxury of considering as many alternative plans as possible to find the best.

Even with these added "knobs" to adjust the size of the search space, better search algorithms and heuristics are needed for the "worst case" scenario of a star query to be optimized interactively. A number of promising heuristics have recently been compared experimentally by Swami [SWAM 89b]. The AI folks seem enamored of the A* algorithm, which is effectively Branch and Bound, a very robust technique that I have tried to couple with our "bottom up" approach of optimization. It requires a tight lower bound to prune partial plans, something that appears to be virtually impossible to accomplish with the dramatic differences in cost functions for various join methods, although Yoo and Lafortune have succeeded for semijoins alone [YOO 88]. The need for an initial feasible solution in Branch and Bound led me to the development of a "greedy" heuristic that is currently used by the optimizer of the SQL DBMS in OS2/ Extended Edition [IBM 88]. Starting with the query graph, this algorithm evaluates all the ways of joining any two tables, and picks the cheapest. The nodes of those two tables are now collapsed into a single

node, and the cost of all alternative join methods for each join corresponding to an edge leading into the collapsed node are re-evaluated. We continue to choose the cheapest join and collapse the joined nodes until only one node remains. Swami's "augmentation" method [SWAM 89b] is similar, but uses a simple measure such as the selectivity of the join predicate to pick the next join, whereas we perform a more thorough evaluation of the costs of each join.

Interaction with Data Placement (Database Design)

This is another topic that has been around for a long time and so appears moribund. However, the added complexity of databases distributed over heterogeneous hardware (workstations, servers, and hosts) connected by heterogeneous links (LANs and WANs), complex objects (particularly when objects can share sub-objects, rather than properly own them), and (again!) recursion raise interesting problems about how best to place data, and how best to exploit that placement when selecting a plan for execution. In Starburst, we haven't yet addressed these problems, largely because we intend to support objects as a generalization of views on tables, for which traditional optimization still applies.

Optimizer Learning

Here's a wide open topic with real sex appeal, yet whose surface has hardly been scratched! Current SQL compilers statically bind a plan in an "access module" to each query, until the plan is invalidated by the disappearance of some object on which it depends (e.g. a table referenced in the query). The SQL/400 product also re-optimizes whenever a new index is added that might benefit the query. And Graefe and DeWitt have suggested weighting the choice of the next transformational rule to apply during optimization based upon how often its transformation has improved the estimated cost over all applications since the optimizer was created [GRAE 87a]. However, as far as I can determine, no one has investigated how to make the optimization process learn from previous *executions* of the same (or related!) queries. It sounds so easy and obvious, but making it really work is much more difficult, largely because writing any performance data to the access module precludes concurrent users from doing likewise. We are investigating various designs for getting around this problem.

Things We *Aren't* Investigating

It's tempting but impractical in a wish list such as this to list a wide variety of topics "under investigation". Bruce Lindsay has often observed that listing what is *excluded* can be more informative than listing what is *included*. Hence, here is a partial (and intentionally controversial) list of topics in which I have little or no interest, along with assorted biases:

- Studies to find "the best join method" for all situations, particularly a YAHOO (Yet Another Hash-join OptimizatiOn) that assumes that both tables fit into memory. For any such uniformly optimal join method, I claim that I can always construct a database and a query for which another join method runs better, and our customers continue to insist upon building databases that don't *quite* fit into those massive and cheap main memories. *However*, we *have* implemented a variety of the Grace join[2] [KITS 83] in Starburst. And we *are* investigating what execution strategies we need for larger main memories

---

2 The Grace join was implemented primarily to test some theories about better communication between the buffer manager and the optimizer [CHOU 85]. Hence it was chosen primarily because it was easier to implement and less sensitive to errors in the optimizer's estimates of the number of buckets vs. the number of buffers, even though DeWitt et al. have shown the Hybrid Hash Join to be uniformly superior in performance [DEWI 84].

(e.g., sequential pre-fetch, as is done in IBM's DB2 product) and the impact these have on the hit ratios and cost equations. Relatively little has been done on "industrial grade" DBMSs when the *entire* database fits in main memory [LEHM 86], [BITT 87], [SWAM 89a], and we hope that their cost equations may be simplified enough to develop better dominance relationships.

- Cost equations. How often have you stolen the cost equations for your optimizer from someone else's paper? The scarce resources, important parameters, and level of detail are unlikely to be the same on any two machines, so why bother?

- The best ordering of semijoins (as in [YOO 88]). Although the Starburst Plan Generator can produce plans with semijoins, given the appropriate STARs, previous experiments [MACK 86] make me skeptical that they will generally be efficient except when the query requests a very wide column that is too "heavy" to lug around while doing joins.

- Parallel join algorithms that look great for 2-table joins because they just happen to be horizontally partitioned (uniformly, of course!) on the join column, but result in shipping every tuple of the result to another site in order to partition on the join column of the next join. Any paper describing a parallel join algorithm that isn't illustrated with a 3-table join (on different join columns!) is not worth reading.

- "SQL as a datatype" (with apologies to Mike Stonebraker [STON 84]). It's a very elegant idea, but I remain skeptical of its practicality due to the potential for a single update near the root to cause a cascade of re-optimizations and re-cacheing of results.

Bibliography

[BITT 87] D. Bitton, M.B. Hanrahan, and C. Turbyfill, Performance of Complex Queries in Main Memory Database Systems, *Procs. of 3rd Intl. Conf. on Data Engr.* (Los Angeles, 1987) pp. 72—81.

[CHOU 85] H.-T. Chou and D.J. DeWitt, An Evaluation of Buffer Management Strategies for Relational Database Systems, *Procs. of 11th Intl. Conf. on Very Large Data Bases (VLDB)* (Stockholm, August 1985) pp. 127—141.

[CHRI 83] S. Christodoulakis,, Estimating Record Selectivities, *Info. Systems* **8,2** (1983) pp. 105—115.

[DAYA 88] U. Dayal, Active Database Management Systems, *Procs. of 3rd Intl. Conf. on Data and Knowledge Bases* (Jerusalem, June 1988) pp. 150—169.

[DEWI 84] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood, Implementation Techniques for Main Memory Database Systems, *Procs. of ACM-SIGMOD* (1984) pp. 1—8.

[FREY 87] J.C. Freytag, A Rule-Based View of Query Optimization, *Procs. of ACM-SIGMOD* (San Francisco, May 1987) pp. 173—180. Also available as IBM Research Report RJ5349, San Jose, CA, October 1986.

[GRAE 87a] G. Graefe and D.J. DeWitt, The EXODUS Optimizer Generator, *Procs. of ACM-SIGMOD* (San Francisco, May 1987) pp. 160—172.

[GRAE 87b] G. Graefe, Rule-Based Query Optimization in Extensible Database Systems, *Ph.D. Thesis* (University of Wisconsin, Madison, WI, August 1987).

[GRAE 87c] G. Graefe, Selectivity Estimation Using Moments and Density Functions, *Tech. Report #CS/E 87-012* (Oregon Graduate Center, Beaverton, OR, November 1987).

[GRAE 89] G. Graefe, The Stability of Query Evaluation Plans and Dynamic Query Evaluation Plans, *Procs. of ACM-SIGMOD* (Portland, OR, May 1989 (to appear)).

[HASA 88] W. Hasan and H. Pirahesh, Query Rewrite Optimization in Starburst, *IBM Research Report RJ6367* (San Jose, CA, August 1988).

[IBM] IBM Corp., Index Key Range Estimator, *U.S. Patent #4774657.*

[IBM 88] IBM Corp., Heuristic Method for Joining Relational Data Base Tables, *IBM Technical Disclosure Bulletin* **30,9** (February 1988) pp. 8—10.

[KITS 83] M. Kitsuregawa, H. Tanaka, T. Moto-oka, Application of Hash to Data Base Machine and Its Architecture, *New Generation Computing* **1** (1983) pp. 63—74.

[LEE 88] M. Lee, J.C. Freytag, and G.M. Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *Procs. of 14th Intl. Conf. on Very Large Data Bases (VLDB)* (Long Beach, August 1988) pp. 218—229. Also available as IBM Research Report RJ6125, San Jose, CA, March 1988.

[LEHM 86] T. Lehman and M. Carey, Query Processing in Main Memory Database Systems, *Procs. of ACM-SIGMOD* (Washington, D.C., 1986) pp. 239—250.

[LOHM 88] G.M. Lohman, Grammar-Like Functional Rules for Representing Query Optimization Alternatives, *Procs. of ACM-SIGMOD* (Chicago, May 1988) pp. 18—27. Also available as IBM Research Report RJ5992, San Jose, CA, December 1987.

[LYNC 88] C.A. Lynch, Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values, *Procs. of 14th Intl. Conf. on Very Large Data Bases* (Long Beach, September 1988) pp. 240—251.

[MACK86] Mackert, L. and G. Lohman, R* Optimizer Validation and Performance Evaluation for Distributed Queries, *Procs. of 12th Intl. Conf. on Very Large Databases* (Kyoto, August 1986).

[MURA 88] M. Muralikrishna and D.J. DeWitt, Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries, *Procs. of ACM-SIGMOD* (Chicago, May 1988) pp. 28—36.

[ONO 88] K. Ono and G.M. Lohman, Extensible Enumeration of Feasible Joins for Relational Query Optimization, *IBM Research Report RJ6625* (San Jose, CA, Dec. 1989).

[PIAT 84] G. Piatetsky-Shapiro, C. Connell, Accurate Estimation of the Number of Tuples Satisfying a Condition, *Procs. of ACM-SIGMOD* (1984) pp. 256—276.

[SELI 79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, Access Path Selection in a Relational Database Management System, *Procs. of ACM-SIGMOD* (1979) pp. 23—34.

[STON 84] M. Stonebraker, E. Anderson, E. Hanson, and B. Rubenstein, QUEL as a Datatype, *Procs. of ACM-SIGMOD* (1984) pp. 208—214.

[SWAM 89a] A. Swami, A Validated Cost Model for Main Memory Databases, *Procs. of ACM-SIGMETRICS Conf. on Measurement and Modeling of Comp. Sys.* (May 1989 (to appear)).

[SWAM 89b] A. Swami, Optimization of Large Join Queries: Combining Heuristics with Combinatorial Techniques, *Procs. of ACM-SIGMOD* (Portland, OR, May 1989 (to appear)).

[VALD 87] P. Valduriez, Join Indices, *ACM Trans. on Database Systems* **12,2** (June 1987) pp. 218—246.

[VAND 86] B.T. Vander Zanden, H.M. Taylor, and D. Bitton, Estimating Block Accesses when Attributes are Correlated, *Procs. of 12th Intl. Conf. on Very Large Data Bases* (Kyoto, September 1986) pp. 119—127.

[WONG 76] E. Wong and K. Youssefi,, Decomposition -- a Strategy for Query Processing, *ACM Transactions on Database Systems* **1,3** (September 1976) pp. 223—241.

[YOO 88] H. Yoo and S. Lafortune, An Intelligent Search Method for Query Optimization by Semijoins, *Tech. Report #CRL-TR-10-88* (Comp. Res. Lab., Univ. of Michigan, Ann Arbor, MI, September 1988).

Five hard problems in query optimization

Patricia G. Selinger
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
(Pat at IBM.COM)

Database technology is at least one generation behind the field of programming languages. This is especially true of query optimization, and is the result of starting two decades later and being a more difficult area. I have listed here what in my opinion are the five hardest problems in query optimization. The reader is warned that these are basically problem descriptions, accompanied in some cases with some possible solution approaches. They are ***not*** solutions; my intent is to inspire more research in these areas.

Problem 1:

Adjustable cost objectives. Most of the basic research in optimization establishes as its cost objective the minimization of the total query cost. The early query optimizers used either total number of expected I/O's or a parameterized combination of CPU and I/O's. Later on, for distributed systems, the cost objective was to minimize the total number of messages or a parameterized combination of total CPU, I/O, and messages. A few papers suggested using minimum response time rather than total cost, but that was not generally adopted.

If you ask a user what cost objective to use, the answer will be "none of the above". The answer will likely be in terms of a case statement: "For application A, I want a fast answer; for application B, I don't care how you do it so long as I get the answer by tomorrow morning; for application C, I don't want you to do it unless you guarantee that you can get it done over the weekend." If you ask a system administrator, the answer will likely be couched in terms of system load and expected response time for some applications: "give the interactive users of application A a 2 sec. response time" and for other applications the answer will be "don't exceed x% of the budget and never use more than y% of the system."

What does this really mean for optimizer research? It means that we need to take a broader view of the optimization problem than we have in the past. Generally, we need to optimize a set of simultaneous equations including response time, and total cost (in differing units such as time or money), in the presence of constraints on completion time and simultaneous system load. Clearly this involves both optimization time work as well as execution time work. That in turn, implies a special relationship between the query optimizer and the execution time parts of the system such as the scheduler and dispatcher.

Clearly this is a nearly impossible (and not well-defined) problem, and is not going to be solved by optimizers next week. What are some reasonable solution approximations? What input parameters can such optimizers expect (or demand) from the database statistics, operating system, etc.? How might users and system administrators specify what constraints and cost objectives they want each application to have?

Problem 2:

Time-varying resources. This problem is closely related to the issues raised in problem 1 above, but takes the complexity one step further. It can also be treated separately from the flexibility and generality needed to solve problem 1. The situation can be described most easily with an example. Suppose a decision support query examines a lot of data, and the user needs the answer by 9AM tomorrow morning. If the query starts at 4PM, the system administrator might choose the following resource utilization and plans:

The first two hours of execution should be a single process at low priority (to avoid delays for interactive users). From 6PM to 10PM, up to 30% of the CPU can be utilized for this query with no more than 10 processes. After 10PM until the query is completed, there are no resource constraints -- unlimited parallel execution is permitted.

This is a resource allocation and scheduling problem that can occur in many domains. It may or may not have a feasible solution. These resource constraints effectively limit the query optimizer's solution search space (repertoire of feasible plans). While finding the optimal plan is NP-complete, there may be non-trivial cases where efficient search can identify several "pretty good" plans.

I'd like to issue an open invitation to operations research folks to work with the database community in attacking both problem 1 and problem 2.

Problem 3:

Heterogeneous distributed optimization. In the first wave of heterogeneous distributed systems, we researchers punted on the issue of global optimization. Most systems relied on global (manually established) catalogs containing global to local schema mappings. These were used to identify which system should receive translated requests and how to translate them. Individual participating databases would receive requests in their "native" interface, possibly with the assistance of a layer of global function sitting on top of the local DBMS.

On the other hand, homogeneous distributed systems aimed at the ultimate global optimization, using intimate knowledge of the access paths and costs of the local DBMSs and relying on the fact that the participating databases were (nearly) identical in function.

Clearly there is a middle ground between these two models and sets of technology. What could be done in heterogeneous distributed optimization when the database language and functions are (nearly) identical, but the systems are heterogeneous? By heterogeneous systems, one can visualize different optimizers, different access paths, different costs, different join repertoires, and so on. Let's call these systems "semi-heterogeneous". Can we invent semi-heterogeneous optimization techniques that produce the performance achievable with homogeneous distributed systems? One approach is a "super-optimizer" that knows the union of all the techniques of each local DBMS optimizer. Another approach is a cooperative decision among peer optimizers. How can we have our cake and eat it too -- that is, when a DBMS instance finds itself in a homogeneous situation, distributed requests should achieve (near) optimal performance, and when in a semi-heterogeneous situation, optimization should produce "as good as it can" plans?

Problem 4:

Optimizing for parallel systems. Because parallelism adds so many additional degrees of freedom, the exhaustive search technology that some systems today use successfully will no longer work. A number of proposals have been made to use rules, simulated annealing, branch and bound, ... I have included this issue because in my opinion more work is needed to evaluate and compare these competing technologies and also to invent new ones. The problem is not yet "solved".

Problem 5:

Borrowing from programming language optimizations. Because we want to have optimizers that search efficiently (but not exhaustively) and are extensible, the trend today is to separate the optimizer search engine from the costs and the target repertoire of plans. If we look at programming languages as a more mature technology, we find that they have approached this issue differently. After making an initial cut at compilation, optimizing compilers dig into a repertoire of peephole optimizations to improve its results, making multiple passes through the output. They use loop unrolling, code migration, common subexpression elimination, procedure integration, and so on to modify and improve their initial output. Many of these techniques also apply to database plans in a relatively straightforward way.

Furthermore, we understand many database specific modifications and improvements (predicate pushdown, subquery to join transformations). Today, some systems apply these separately from the search engine enumeration of feasible plans, often as a preprocessing step before access path selection.

The problem that needs to be examined by the database optimizer community is twofold. First, identify peephole postprocessing optimizations that can be stolen from programming language compilers, and invent more. Second, examine whether we can do better job of integrating query rewriting, search space enumeration, cost estimation, and peephole post processing. Are there better ways of combining these steps more intimately without abandoning the extensibility and modularity benefits from isolating the search engine from the search strategies?

Bibliography

[CHOU 85] H.-T. Chou and D.J. DeWitt, An Evaluation of Buffer Management Strategies for Relational Database Systems, *Procs. of 11th Intl. Conf. on Very Large Data Bases (VLDB)* (Stockholm, August 1985) pp. 127–141.

[CHRI 83] S. Christodoulakis,, Estimating Record Selectivities, *Info. Systems* **8,2** (1983) pp. 105–115.

[FREY 87] J.C. Freytag, A Rule-Based View of Query Optimization, *Procs. of ACM-SIGMOD* (San Francisco, May 1987) pp. 173–180. Also available as IBM Research Report RJ5349, San Jose, CA, October 1986.

[GRAE 87a] G. Graefe and D.J. DeWitt, The EXODUS Optimizer Generator, *Procs. of ACM-SIGMOD* (San Francisco, May 1987) pp. 160–172.

[GRAE 87b] G. Graefe, Rule-Based Query Optimization in Extensible Database Systems, *Ph.D. Thesis* (University of Wisconsin, Madison, WI, August 1987).

[GRAE 87c] G. Graefe, Selectivity Estimation Using Moments and Density Functions, *Tech. Report #CS/E 87-012* (Oregon Graduate Center, Beaverton, OR, November 1987).

[GRAE 89] G. Graefe, The Stability of Query Evaluation Plans and Dynamic Query Evaluation Plans, *Procs. of ACM-SIGMOD* (Portland, OR, May 1989 (to appear)).

[HASA 88] W. Hasan and H. Pirahesh, Query Rewrite Optimization in Starburst, *IBM Research Report RJ6367* (San Jose, CA, August 1988).

[IBM] IBM Corp., Index Key Range Estimator, *U.S. Patent #4774657.*

[IBM 88] IBM Corp., Heuristic Method for Joining Relational Data Base Tables, *IBM Technical Disclosure Bulletin* **30,9** (February 1988) pp. 8—10.

[KITS 83] M. Kitsuregawa, H. Tanaka, T. Moto-oka, Application of Hash to Data Base Machine and Its Architecture, *New Generation Computing* 1 (1983) pp. 63—74.

[LEE 88] M. Lee, J.C. Freytag, and G.M. Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *Procs. of 14th Intl. Conf. on Very Large Data Bases (VLDB)* (Long Beach, August 1988) pp. 218—229. Also available as IBM Research Report RJ6125, San Jose, CA, March 1988.

[LOHM 88] G.M. Lohman, Grammar-Like Functional Rules for Representing Query Optimization Alternatives, *Procs. of ACM-SIGMOD* (Chicago, May 1988) pp. 18—27. Also available as IBM Research Report RJ5992, San Jose, CA, December 1987.

[LYNC 88] C.A. Lynch, Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values, *Procs. of 14th Intl. Conf. on Very Large Data Bases* (Long Beach, September 1988) pp. 240—251.

[MACK86] Mackert, L. and G. Lohman, R* Optimizer Validation and Performance Evaluation for Distributed Queries, *Procs. of 12th Intl. Conf. on Very Large Databases* (Kyoto, August 1986).

[ONO 88] K. Ono and G.M. Lohman, Extensible Enumeration of Feasible Joins for Relational Query Optimization, *IBM Research Report RJ6625* (San Jose, CA, Dec. 1989).

[PIAT 84] G. Piatetsky-Shapiro, C. Connell, Accurate Estimation of the Number of Tuples Satisfying a Condition, *Procs. of ACM-SIGMOD* (1984) pp. 256—276.

[SELI 79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, Access Path Selection in a Relational Database Management System, *Procs. of ACM-SIGMOD* (1979) pp. 23—34.

[SWAM 89b] A. Swami, Optimization of Large Join Queries: Combining Heuristics with Combinatorial Techniques, *Procs. of ACM-SIGMOD* (Portland, OR, May 1989 (to appear)).

[VALD 87] P. Valduriez, Join Indices, *ACM Trans. on Database Systems* **12,2** (June 1987) pp. 218—246.

[VAND 86] B.T. Vander Zanden, H.M. Taylor, and D. Bitton, Estimating Block Accesses when Attributes are Correlated, *Procs. of 12th Intl. Conf. on Very Large Data Bases* (Kyoto, September 1986) pp. 119—127.

[WONG 76] E. Wong and K. Youssefi,, Decomposition -- a Strategy for Query Processing, *ACM Transactions on Database Systems* **1,3** (September 1976) pp. 223—241.

[YOO 88] H. Yoo and S. Lafortune, An Intelligent Search Method for Query Optimization by Semijoins, *Tech. Report #CRL-TR-10-88* (Comp. Res. Lab., Univ. of Michigan, Ann Arbor, MI, September 1988).

Discussion Issues for Query Processing

Arnon Rosenthal

This document contains three short position statements about issues in query processing: query optimization for a federated, object-oriented system; outerjoins; representations and rigor.

1. Query Processing for a Federated, Object-Oriented System

There has been substantial interest in support for "interoperability", to enable separately-developed tools to communicate. Many researchers believe that a federated, object-oriented system can best provide the global environment. Such a system should also provide some traditional database services: definitions of virtual objects (views), and queries to large collections.

We have been considering an architecture that is built around an object management system (OMS) that supplies an object-oriented data mdoel and invocation of arbitrary functions (including data retrievals). All data resides in data servers developed separately from the OMS; each site will have its own array of DBMSs, file managers, and application-managed repositories. User-defined datatypes must be supported. Finally, the system includes a simple language for writing expressions (e.g., selection followed by join).

Current systems split such a world into three separately-optimized domains:

A. *Programming language expressions*: User-defined functions belong in this world. Typical optimizations include constant propagation, replacing a procedure call by in-line code (permitting further optimization), and removal of redundant operations.

B. *Query language expressions*: The query language consists of operators on collections of objects. It is subject to optimizations such as permuting selections, projections, and joins.

C. *Access to data in servers*: It is rarely necessary to retrieve an entire collection of objects from a server. Instead, an incoming expression may be transformed to push much of it into the server. A strong server (e.g., a SQL DBMS) may be able to execute a large subexpression internally (using indexes, joins, etc.). Also, repeated requests to a single server can often be sent as a batch.

It is undesirable to build three separate optimizers, or to have all optimization confined within the individual domains. A rule-based optimizer seems like a natural approach to building a unified, extensible optimizer, but the task of controlling such a wide variety of rules is daunting.

Extensibility: The effort of attaching each new data server must be minimized. We have a solution such that:

Xerox Advanced Information Technology. 4 Cambridge Center. Cambridge MA 02142 arnie@xait.xerox.com (617)-499-4462

- It is easy to make data in a server available for object-at-a-time navigation. (In engineering, navigation is the main access mode and queries are rare.). Set-oriented queries will also be supported, but inefficiently, by retrieving entire collections from the server and then using OMS query operators.
- Optimizing transformations can be provided incrementally, to replace a query subexpression by a call to the server that holds the data (if the server has the necessary capability). The installation or the vendor of a data server can also invest incrementally in data administration tools to keep the global metadata consistent with the metadata within the server.

In this environment, SQL is awkward as the interface language. It is hard to extend (and then becomes nonstandard), and awkward to manipulate. Instead, we plan to manipulate operator-graphs, and to provide small translators to and from SQL.

2. Simple Outerjoins

Outerjoins present substantial problems in query specification and optimization, because associativity, commutativity, etc. are lacking. However, we conjecture that there is a useful class of queries for which outerjoins are "well-behaved" (i.e., reassociate freely). For this class, language design and query optimization will be much simpler than for general queries involving outerjoins.

Query Specification: In an algebraic language, an outerjoin operator causes no confusion -- the semantics are simply to evaluate each operator as shown in the tree. But it is difficult to add outerjoins to a calculus language or to SQL, because the user does not explicitly parenthesize operators in a multiway join.

There appear to be patterns of outerjoins such that the result will be the same for all legal parenthesizations of the operators. To see which queries are well behaved, we extend the *relation-adjacency* graph of System R to hold outerjoin information. The graph consists of one node for each relation, undirected edges for regular joins, and directed edges for outerjoins. Each edge is labelled with a join predicate. Selections on one relation will be considered later. For simplicity, we'll assume that the graph is connected.

A *parenthesization* is a binary tree whose leaves are the relations in the query; each internal node corresponds to a set of graph edges that connect nodes in the left and right subtrees, i.e., to a set of join predicates. The operator tree *immediately derived* from an association is defined as follows: If a tree node corresponds to a set of undirected edges, perform an ordinary join (using the conjunction of their predicates). If it corresponds to a single directed edge, perform an outerjoin. Otherwise the operation is undefined. Operator trees are also called *expressions*. It is critical that a good parenthesization be chosen, since the cost of two parenthesizations can differ enormously.

In the examples below, --> denotes left-outerjoin, and --- denotes regular join. Suppose that the join predicates link the pairs (R1, R2) and (R2, R3). Each of Examples 1.1-1.3 shows a graph and two expressions immediately derived from different associations. In 1.1 and 1.2 the expressions compute the same result (which may be considered the result of the graph). In 1.3 the graph's result is not well defined, since the two expressions compute different results -- the query language must state which association is desired.

Example 1.1

```
Graph:                    R1 --> R2 --> R3
Expressions:     R1 --> (R2 --> R3) = (R1 --> R2) --> R3
```

Example 1.2

```
Graph:                    R1 --- R2 --> R3
Expressions:     R1---(R2 --> R3) = (R1 ---R2) --> R3
```

Example 1.3

```
Graph:                    R1 --> R2 --- R3
Expressions:  R1 -->(R2 ---R3)  is not equal to (R1 --> R2) ---R3
```

Conjecture: A graph is called *simple* if the directed edges form an outward-directed forest, and no edge of this forest is directed toward an undirected edge. We conjecture that for every join/outerjoin/project query whose graph is simple, the result is well-defined. That is, all immediately-derived operator trees compute the same result. Furthermore, for any nonsimple graph, there appears to be a query that cannot be reassociated.

Examples 1.1 and 1.2 have simple graphs. Example 1.3 is not simple.

Consequences of the conjecture:

- *Language*: a language that expresses only queries with simple graphs need not be concerned with parenthesization [1]
- *Optimization*: An ordinary relational optimizer can be extended relatively easily to handle outerjoins with simple graphs. [2]

adding Selections to the stew: When a query graph has selections or 3-way joins, we consider that all selections on a relation must be performed before any outerjoins. If a query comes in as an operator tree, selections need to be moved down before the query can be expressed as a simple operator graph. We consider only selection predicates that fail whenever a referenced attribute is null. Then we push the selection [attribute≠null] downward to each relation referenced in the offending selection; as a result, the outerjoin on that relation is converted to an ordinary join.

```
Example:  Select(R1.A+R2.B+R3.C <10) [(R1 --> R2) --> R3] =
          Select(R1.A+R2.B+R3.C <10) [R1 ---  R2  --> R3]
```

Applications: In addition to 1-sided outerjoins, there are many other important operators that

---

[1] We have seen at least one proposal for a SQL extension all of whose outerjoins had simple graphs. Relation-lists used "comma" to mean join and "arrow" to mean outerjoin, and the use of arrow was restricted to nested tables.

[2] To see this, note that current optimizers generate parenthesizations for queries with ordinary joins; with simple outerjoins, they determine whether each join node should be a regular join, outerjoin, or disallowed. (For non-simple graphs, they will need additional logic to add operations to ensure the correctness of the result [Rosenthal&Reiner, VLDB84; Dayal, VLDB87]. Some other extensions are also needed. The cost model and the join implementations (e.g., nested loops and merging) must be extended. Selections are handled by query transformations.

combine information from two relations. For example, two-sided outerjoins are important in defining a view that merges information from two databases describing overlapping sets of real-world entities. Other classes include semijoins, universal joins, and nested subqueries -- all combine information from multiple sources. It is important to have intermediate-level concepts and lemmas, if we are to avoid subtle errors in proposed transformations.

3. Representations and Rigor in Query Optimization

There is a strong case for using an an operator graph (or "strategies graph" for multiple strategies) as the basis for all query optimization (including view substitution, query modification, and strategy generation). The advantages of such approaches are: a graph's output is well defined (as long as its operators are defined); the graph's "language" is extensible (by adding new operators); it shows the order of operations (so a transformation is actually affecting execution). For even greater generality, instead of hard-wiring the operator to the node we might let operator nodes be LISP-like S-expressions. This provides greater visibility for expressions (e.g., predicate expressions) within the operators.

Note that operators may be large -- at one phase of optimization, operators in Starburst correspond to Select/From/Where queries. [3] Three refinements are needed before operator graphs become suitable for all optimization:

A. *Representations for SETS of strategies*: The strategies generated by an optimizer have much in common, and this needs to be exploited. In many current optimizers, this exploitation is in the implementation for Select/Project/Join queries, but is unavailable for other purposes.

A strategies graph [Rosenthal&Reiner, Database Engineering 1982] shows a set of alternative strategies, while representing common subexpressions only once. (This representation has also been called a strategy space or And/Or graph). [4] The strategies graph is typically much smaller than maintaining a set of strategies. At high levels of abstraction, the cost of producing it explicitly seems small, since there are many fewer nodes that at the physical implementation level. One can then freely elaborate parts of it to greater levels of detail.

B. *A single formalism at all levels*: Operator trees may contain low-level, executable operators (e.g., scan via index, merge-join), and large operators (e.g., nodes of Starburst's QGM can include Select/From/Where expressions), and high-level operators (e.g. a recursion operator).

It seems undesirable to give a completely separate conceptual treatment to high-level operators versus operators having direct implementation. First, different systems might differ in what operators are directly implemented. Second, it might be reasonable to use similar techniques at both levels (e.g., transformations, verification, tools to identify referenced attributes). Instead, we might want to build an optimizer as a collection of cooperating experts, each with expertise in some phase

[3] Starburst passes more than just an operator tree among optimizer components -- it includes derived information such as adjacency graphs, costs, etc. This appears a good software interface, but may not need to be visible for theoretical purposes.

[4] Our formulation included non-executable nodes showing several alternatives -- G. Lohman and G. Graefe have pointed out that these nodes could instead be executable "Choice" operators.

of optimization. [However, I don't know whether we can effectively control the interaction of all these experts in an extensible system.]

C. *Expressing all operations algebraically*: Currently, optimization of nested subqueries is done in a world that is rather separate from the rest of the optimizer, and where there are few lemmas as building blocks. Perhaps as a consequence, incorrect algorithms have been published (and no doubt implemented). An "iteration" annotation on a query graph may be a useful formalism (as in Starburst), but one needs to define its semantics very carefully with respect to the rest of the query.

Two other ways of handling iteration deserve mention. A good one, now becoming popular, is the use of Stream objects, and operators that produce them. An alternative formalization that is more general is the "Apply-Append" operator [Manola&Dayal, OODMS87]. Apply-append allows iteration to be expressed as an ordinary operator that can be placed in an operator graph. It coexists nicely with a relational algebra approach to optimization. It provides a useful intermediate level in defining the semantics of nested subqueries. Formally, let f() be any function defined on tuples of relation R1. Then apply-append(R1, f) is defined to produce a relation whose tuples are of the form [t1, f(t1)], for each tuple of t1.

Verification: It seems reasonable to try to verify an optimizer's algorithms (at least, informally), and to specify what invariants must be preserved by each module of the optimizer. One way to think about such issues is to regard the optimizer as applying productions that add or remove alternatives to a strategies graph. These productions seem to be of several types:

A. *Equivalence*: For some subgraph, add an equivalent structure as an alternative. This kind of transformation can be justified by algebraic identities

B. *Self-describing subgraphs*: For some subgraph, add a non-equivalent structure. For example, add a subgraph that "implements" the input subgraph, but imposes additional properties (e.g., sort order, cost estimates). Or add a subgraph that produces a result that we suspect will be useful later (e.g., a relation, clustered in an interesting way). This kind of production is justified by examining the operator semantics (expressed by pre- and post-conditions), and comparing these conditions with the properties attached to the operator's input and output nodes.

C. *Search*: Transformations that delete subgraphs that appear suboptimal. These are justified by dynamic programming, branch and bound, bounds on the possible error, or crossed fingers.

Research directions in the Optimization of a Logic Based Language

R. Krishnamurthy
MCC, 3500 Balcones Center Dr., Austin, TX, 78759

The Logic Data Language, LDL, combines the expressive power of a high-level logic-based language (e.g., Prolog) with the non-navigational style of relational query languages, where the user need only supply a query (stated logically), and the system (i.e., the compiler) is expected to devise an efficient execution strategy for it. Consequently, the query optimizer is delegated the responsibility of choosing an optimal execution -- a function similar to that of an optimizer in a relational database system. The optimizer uses the knowledge of storage structures, information about database statistics, estimation of cost, etc. to predict the cost of various execution schemes chosen from a pre-defined search space, and selects a minimum cost execution.

As compared to relational queries, LDL queries pose a new set of problems which stem from the following observations. First, the model of data is enhanced to include complex objects (e.g., hierarchies, heterogeneous data allowed for an attribute [Z 85]). Secondly, new operators are needed not only to operate on complex data, but also to handle new operations such as recursion, negation, etc. Thus, the complexity of data as well as the set of operations emphasize the need for new database statistics and new estimations of cost. Finally, the use of evaluable functions (i.e., external procedures), and function symbol [TZ 86] in conjunction with recursion, provides the ability to state queries that are *unsafe* (i.e., do not terminate). As unsafe executions are a limiting case of poor executions, the optimizer guarantees the choice of a safe execution.

The knowledge base consists of a *rule base* and a *database*. A rule may be *recursive*, in the sense that the definition in the body may depend on the predicate in the head, either directly by reference or transitively through a predicate referenced in the body. A set of predicates that are mutually recursive is said to be a *recursive clique*.

In a departure from previous approaches to compilation of logic [KT 81, U 85, N 86], we make our optimization query-specific. A predicate P1(c,y), (in which c and y denote a bound and unbound argument respectively), computes all tuples in P1

that satisfies the constant, c. A *binding* for a predicate is the bound/unbound pattern of its arguments, for which the predicate is computed. A predicate with a binding is called a *query form* (e.g., P1(c,y)?). We say that the optimization is query-specific because the algorithm is repeated for each such query form. For instance, P1(x,y)? will be compiled and optimized separately from P1(c,y)?. Indeed the execution strategy chosen for P1(c,y)? may be inefficient (or even unsafe) for P1(x,y)?.

We formally define the optimization problem as follows: "Given a query Q, an execution space E and a cost model defined over E, find an execution in E that is of minimum cost." We discuss the research directions in the context of this formulation of the problem.

Model of Execution

An execution models the relevant properties of the actual query processing by the underlying engine. The relevant properties should include all parameters that are to be chosen by the optimizer (e.g., create index, join method) as well as the information that is needed to evaluate the cost of the execution (e.g., projected attributes, duplicate elimination). The model of an execution consists of two types of information: 1) structure, and 2) annotation (to the structure). The optimal choice of structure for a given query, typically, requires the exhaustive search of an exponential space of executions, whereas the annotation represents those information that can be greedily chosen for a given structure.

The structure of an execution is represented by an AND/OR graph. This representation is similar to the predicate connection graph [KT 81], or rule graph [U 85], except that we give specific semantics to the internal nodes as described below. In keeping with our relational algebra based execution model, we map each AND node into a *join* and each OR node into a *union*. Recursion is implied by an edge to an ancestor. A *contraction* of a clique is the extrapolation of the traditional notion of an edge contraction in a graph. An edge is said to be *contracted* if it is deleted and its ends (i.e., nodes) are identified (i.e., merged). A clique is said to be *contracted* if all the edges of the clique are contracted. Intuitively, the contraction of a clique consists of replacing the set of nodes in the clique by a single node and associating all the edges in/out of any node in the clique with this new node.

The annotation provides all other information that are needed to model the execution. Intuitively, a parameter or property is modeled as an annotation if, for a given structure, the optimal choice of that information can be greedily chosen. For example, given the ordering of the joins for a conjunctive query, the choice of access methods, creation of indices, and pushing of selection are examples of choices that can be greedily decided. On the other hand, the pushing of selection into a recursive clique is not a property that can be greedily chosen.

The space of executions over which the optimization problem is defined, is characterized by the of valid structures for a given query and database, and the associated optimal annotations. This is the space over which the search for optimal execution is conducted.

Much of the research in the context of recursive query processing has been focused in defining the valid executions that enable the selections to be pushed into recursive cliques. Some important results can be found in [BMSU 85, BeRa, HeNa, KiLo, SaZ1, SaZ4, Vie]. This set of references is in no means complete as this topic deserves a treatise in its own right. Much less research has been conducted in the topic of pushing of projections and the possible set of resulting executions[RBK88,KiLo]. Even less, if any, research has been done in defining the execution space in the context of programs with set unification [ShTZ], updates[NK88], declarative cut[KN88], and other such constructs in logic programming.

Search Strategy

The traditional approach in commercial DBMS has been to use exhaustive search over the execution space. Unfortunately, this becomes unfeasible for most queries of interest in LDL. Therefore, a renewed interest in devising a better search strategy is in progress [KBZ86, SmGe, IW87, SG88, S89]. The search strategies can be classified into three catagories: exhaustive, stochastic, polynomial. These are described below.

The traditional DBMS approach to using exhaustive search is to use the dynamic programming algorithm proposed in [Sell 79]. It is well known that even this is rendered useless if there is a join of 15 relations. In [KrZa] we propose an exhaustive search for optimizing LDL programs.

Another approach to searching the large search space is to use a stochastic algorithm. Intuitively, the minimum cost execution can be found by picking, randomly, a "large" number of executions from the execution space and choosing the minimum cost execution. Obviously, the number of executions that need to be chosen approaches the size of the search space for a reasonable assurance of obtaining the minimum. This number is claimed to be much smaller by using a technique called Simulated Annealing [IW 87]. There have been other variations on this approach [SG88].

Another approach to tackling the large search space is to observe some property of the cost function that results in an efficient search. In [KBZ 86], we presented a quadratic time algorithm that computes the optimal ordering of conjunctive queries when the query is acyclic and the cost function satisfies a linearity property called the Adjacent Sequence Interchange (ASI) property. Further, this algorithm was ex-

tended to include cyclic queries and other cost models. This approach has been extended for nonrecursive queries. But no such attempt has been made to include recursive queries.

In short, we have summarized three generic strategies: exhaustive, quadratic and stochastic. The main trade-offs amongst these strategies is between efficiency (i.e., time complexity) and flexibility. Note that the quadratic strategy is the most efficient, whereas it is least flexible in terms of the possible modifications to cost functions, structure, etc. The design of the LDL optimizer is capable of using multiple strategies interchangeably. The main reason for requiring flexibility in the system is that the system was initially intended as an experimental vehicle since there was no prior experience in the design of an optimizer for a logic language. Thus new ideas may be forthcoming that should be incorporated into the system.

Cost Model

The cost model assigns a cost to each execution, thereby ordering them. Intuitively, the cost of an execution is the sum of the cost of individual operations. In the case of nonrecursive queries, this amounts to summing up the cost for each operation. Therefore, the cost function must be capable of computing the cost of each operation based on the descriptors of the operands. Three major problems are faced in devising such cost functions: 1) computation of the descriptors, 2) estimating the cost of external predicates, 3) safety of recursive queries.

In the presence of nested views, especially with recursion and complex objects, estimating the descriptor for a relation corresponding to a predicate is a very difficult problem. This is further complicated by the fact that logic based languages allow the union of non-homogenous sets of objects. The net effect is that the estimation of the descriptor for any predicate is, in effect, computing the query in an algebraic fashion. That is, the program is executed in the abstract domain instead of the concrete domain. For instance, the *age* attribute may take on values such as 16 in the concrete domain whereas, in the abstract domain, it takes on values such as *integer between 16 to 65*. Obviously, computation in this domain is very difficult and approximations to such computation must be devised that are not only efficient but are also effective. Some work in the context of Prolog has been attempted [DW87, WHD88] and this needs to be extended to the context of LDL.

In LDL, external procedures are treated in an interchangeable manner with any predicate. Intuitively, the external procedure is viewed as an infinite relation satisfying some constraints. Therefore, a concise descriptor of such an infinite relation must be declared in the schema and the cost functions for the operations on these infinite relations must be devised. The abstraction of the approach taken in LDL has been presented in [CGK89]. To our knowledge, this is the only treatise on an op-

timizer dealing with such an integration and this is limited to a subset of LDL. Much more work is needed to include other operations such as updates and the validation of this approach is still lacking.

The cost model must associate an infinite cost for an execution that computes an infinite answer or that never completes. Such *unsafe* queries are to be detected so that the optimizer can avoid choosing them. Obviously, checking for such termination properties is in general undecidable. Sufficient conditions have been proposed [UV85, Z86, KRS88]. In particular, LDL uses the algorithm proposed in [KRS88], which is an enumerative algorithm that exhausts an exponential number of cases. Therefore, it is of interest to devise more efficient algorithms and more comprehensive tests to check for safety.

In summary, we have presented some of the important problems that we have observed in the context of optimizing LDL programs, some of which are currently being pursued.

References:

[ApBW] Apt, K., H. Blair, A. Walker, Towards a Theory of Declarative Knowledge, in *Foundations of Deductive Databases and Logic Programming*, (Minker, J. ed.), Morgan Kaufman, Los Altos, 1987.

[AU 79] Aho, A. and J. Ullman, Universality of Data Retrieval Languages, *Proc. POPL Conf.*, San Antonio, TX, 1979.

[BaBu] Bancilhon, F. and P. Buneman (eds.), *Workshop on Database Programming Languages*, Roscoff, Finistere, France, Sept. 87.

[BMSU85] Bancilhon, F., D, Maier, Y. Sagiv and Ullman, Magic Sets and other Strange Ways to Implement Logic Programs, *Proc. 5-th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pp. 1-16, 1986.

[Ban] Bancilhon, F., Naive Evaluation of Recursively defined Relations, *On Knowledge Base Management Systems*, (M. Brodie and J. Mylopoulos, eds.), Springer--Verlag, 1985.

[BaR] Balbin, I., K. Ramamohanarao, A Differential Approach to Query Optimization in Recursive Deductive Databases, *Journal of Logic Programming*, Vol. 4, No. 2, pp. 259--262, Sept 1987.

[BeRa] Beeri, C. and R. Ramakrishnan, On the Power of Magic, *Proc. 6th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1987.

[Bet1] Beeri, et al., Sets and Negation in a Logic Data Language (LDL1), *Proc. 6th ACM SIGMOD--SIGACT Symp. on Principles of Database Systems*, pp. 269--283, 1987.

[Bet2] Beeri, et al., Bound on the Propagation of Selection in Logic Programs, *Proc. 6th ACM SIGMOD-- SIGACT Symp. on Principles of Database Systems*, 1987.

[BKBR] Beeri, C., P. Kanellakis, F. Bancilhon, R. Ramakrishnan, Bound on the Propagation of Selection into Logic Programs, *Proc. 6th ACM SIGMOD--SIGACT Symp. on Principles of Database Systems*, 1987.

[BR 86] Bancilhon, F., and R. Ramakrishan, An Amateur's Introduction to Recursive Query Processing Strategies, *Proc. 1986 ACM-SIGMOD Intl. Conf. on Mgt. of Data*, pp. 16-52, 1986.

[Boc] Bocca, J., On the Evaluation Strategy of Educe, *Proc. 1986 ACM--SIGMOD Conference on Management of Data*, pp. 368--378, 1986.

[CeGW] Ceri, S., G. Gottlob and G. Wiederhold, Interfacing Relational Databases and Prolog Efficiently, *Expert Database Systems*, L. Kerschberg (ed.), Benjamin/Cummings, 1987.

[Ceta] Chimenti D. et al., An Overview of the LDL System, *Database Engineering Bulletin, Vol. 10, No. 4*, pp. 52--62, 1987.

[Col] Colmerauer, Equations and Inequations in Finite and Infinite Trees, *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 85--99, ICOT, Tokyo, Japan, 1984.

[deSi] deMandreville C. and E. Simon, Modelling Queries and Updates in Deductive Databases *Proc. 1988 VLDB Conference*, Los Angeles, California, August 1988.

[DW86] Debray, S. K., and D. S. Warren, "Automatic Mode inference for Prolog Programs", in Proc. of 1986 Intl. Symp. on Logic Programming, 1986, Salt Lake City, Utah.

[GaDe] Gardarin, G. and C. deMandreville, Evaluation of Database Recursive Logic Programs as Recursive Function Series, *Proc. ACM SIGMOD Int. Conference on Management of Data*, Washington, D.C., May 1986.

[GMN] Gallaire, H.,J. Minker and J.M. Nicolas, Logic and Databases: a Deductive Approach, *Computer Surveys*, Vol. 16, No. 2, 1984.

[GM 82] Grant, J. and Minker J., On Optimizing the Evaluation of a Set of Expressions, *Int. Journal of Computer and Information Science, 11, 3 (1982), 179-189.*

[Har] Harel, D., First-Order Dynamic Logic, *Lecture Notes in Computer Science*, (G. Goos and J. Hartmanis, eds.), Springer Verlag, 1979.

[HeNa] Henschen, L.J., Naqvi, S. A., On compiling queries in recursive first-order databases, *JACM 31, 1, 1984*, pp. 47--85.

[ImNa] Imielinski, T. and S. Naqvi, Explicit Control of Logic Programs Through Rule Algebra, *Proc. 7th ACM SIGMOD--SIGACT Symp. on Principles of Database Systems, pp. 103--116, 1988.*

[IW 87] Ioannidis, Y. E, Wong, E, Query Optimization by Simulated Annealing, *SIGMOD 87*, San Francisco.

[JaCV] Jarke, M., J. Clifford and Y. Vassiliou, An Optimizing Prolog Front End to a Relational Query System, *Proc. 1984 ACM--SIGMOD Conference on Management of Data*, pp. 296--306, 1986.

[KeOT] Kellog, C., A. O'Hare and L. Travis, Optimizing the Rule Data Interface in a KMS, *Proc. 12th VLDB Conference*, Tokyo, Japan, 1986.

[KiLo] Kifer, M. and Lozinskii, E.L., Filtering Data Flow in Deductive Databases, *ICDT'86*, Rome, Sept. 8--10, 1986.

[KoPa] Kolaitis G. P. and C.H. Papadimitriou, Why Not Negation by Fixpoint?, *Proc. 7th ACM SIGMOD--SIGACT Symp. on Principles of Database Systems*, pp. 31--239,1988.

[KBZ 86] Krishnamurthy, R., Boral, H., Zaniolo, C., Optimization of Nonrecursive Queries, *Proc. of 12th VLDB*, Kyoto, Japan, 1986.

[KRS 87] Krishnamurthy, R, Ramakrishnan, R, Shmueli, O., Testing for Safety and Effective Computability, Manuscript in Preparation.

[KrN1] Krishnamurthy and S. Naqvi, Non-Deterministic Choice in Datalog, *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, June 27--30, Jerusalem, Israel.

[KrN2] Krishnamurthy and S. Naqvi, Towards a Real Horn Clause Language, *Proc. 1988 VLDB Conference*, Los Angeles, California, August 1988.

[KrRS] Krishnamurthy, R. R. Ramakrishnan and O. Shmueli, A Framework for Testing Safety and Effective Computability, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 154--163, 1988.

[KrZa] Krishnamurthy, R. and C. Zaniolo, Optimization in a Logic Based Language for Knowledge and Data Intensive Applications, in *Advances in Database Technology, EDBT'88*, (Schmidt, Ceri and Misssikoff, Eds), pp. 16--33, Springer-Verlag 1988.

[KT 81] Kellog, C., and Travis, L. Reasoning with data in a deductively augmented database system, in *Advances in Database Theory: Vol 1*, H.Gallaire, J. Minker, and J. Nicholas eds., Plenum Press, New York, 1981, pp 261–298.

[KuYo] Kunifji S., H. Yokota, Prolog and Relational Databases for 5th Generation Computer Systems, in *Advances in Logic and Databases, Vol. 2* (Gallaire, Minker and Nicolas eds.), Plenum, New York, 1984.

[Li] Li, D. *A Prolog Database System*, Research Institute Press, Letchworth, Hertfordshire, U.K., 1984

[Llo 84] Lloyd, J. W., *Foundations of Logic Programming*, Springer Verlag, 1984.

[M 84] Maier, D., *The Theory of Relational Databases*, (pp. 542–553), Comp. Science Press, 1984.

[Meta] Morris, K. et al. YAWN! (Yet Another Window on Nail!), *Data Engineering, Vol.10, No. 4*, pp. 28--44, Dec. 1987.

[Nai] Naish, L., Negation and Control in Prolog, *Lecture Notes in Computer Science 238*, Springer Verlag 1986.

[NaKr] Naqvi, S. and R. Krishnamurthy, Semantics of Updates in Logic Programming, *Proc. 7th ACM SIGMOD--SIGACT Symp. on Principles of Database Systems*, pp. 251--261, 1988.

[Naq] Naqvi, S. A Logic for Negation in Database Systems, in *Foundations of Deductive Databases and Logic Programming*, (Minker, J. ed.), Morgan Kaufman, Los Altos, 1987.

[Na 86] Naish, L., Negation and Control in Prolog, *Journal of Logic Programming*, to appear.

[NaTs] Naqvi, S. and S. Tsur, *A Logic Language for Data and Knowledge Bases*, MCC Technical Report, 1988.

[Okee] O'keefe, R.A., On the Treatment of Cuts in Prolog Source Level Tools, *Proc. Symposium on Logic Programming*, pp. 68--73, 1985.

[RaBK] Ramakrishnan, R., C. Beeri and Krishnamurthy, Optimizing Existential Datalog Queries, *Proc. 7th ACM SIGMOD--SIGACT Symp. on Principles of Database Systems*, pp. 89--102, 1988.

[Reta] Ramamohanarao, K. et al., The NU-Prolog Deductive Database System, *Database Engineering Bulletin, Vol. 10, No. 4*, pp. 10--19, 1987.

[RLK] Rohmer, J., R. Lescouer and J.M. Kerisit, The Alexander Method ---A technique for the Processing of Recursive Axioms in Deductive Databases, *New Generation Computing, Vol. 4, No. 3*, pp. 273--287,1986.

[S89] Swami, A., "Optiomization of Large Join Queries: Combining Heuristics and Combinatorial Techniques" in Proc. of SIGMOD, Portland OR, 1989.

[SaZ1] Sacc\'{a} D., Zaniolo, C., On the implementation of a simple class of logic queries for databases, *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1986.

[SaZ2] Sacc\'{a} D., Zaniolo, C., Implementation of Recursive Queries for a Data Language based on Pure Horn Logic, *Proc. Fourth Int. Conference on Logic Programming*, Melbourne, Australia, 1987.

[SaZ4] Sacc\'{a} D., Zaniolo, C., Magic Counting Methods, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1987.

[SaZ5] Sacc\'{a} D., Zaniolo, C., Differential Fixpoint Methods and Stratification of Logic Programs, *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, June 2730, Jerusalem, Israel.

[Sel 79] Sellinger, P.G. et. al., Access Path Selection in a Relational Database Management System., *Proc. 1979 ACM-SIGMOD Intl. Conf. on Mgt. of Data*, pp. 23-34, 1979.

[ShTZ] Shmueli, O., S. Tsur and C. Zaniolo, Rewriting of Rules Containing Set Terms in a Logic Data Language (LDL), *Proc. 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, pp. 15--28, 1988.

[SmGe] Smith, D.E. and M.R. Genesereth, Ordering Conjunctive Queries, *Artificial Intelligence, 26,* pp. 171--185, 1985.

[SG88] Swami, A., A. Gupta, "Optimization of Large Join Queries", in Proc. of SIGMOD conference, 1988, Chicago, pp8-17.

[SZ 86] Sacca', D. and C. Zaniolo, The Generalized Counting Method for Recursive Logic Queries, *Proc. ICDT '86 --Int. Conf. on Database Theory,* Rome, Italy, 1986.

[TZ 86] Tsur, S. and C. Zaniolo, LDL: A Logic-Based Data Language,*Proc. of 12th VLDB,* Kyoto, Japan, 1986.

[U 85] Ullman, J. D., Implementation of logical query languages for databases, *TODS,* 10, 3, (1985), 289-321.

[UV 85] Ullman, J.D. and A. Van Gelder, *Testing Applicability of Top-Down Capture Rules,* Stanford Univ. Report STAN-CS-85-146, 1985.

[Ull] Ullman, J.D., *Database and Knowledge-Based Systems,* Computer Science Press, Rockville, Md., 1988.

[V 86] Villarreal, E., *Evaluation of an O(N**2) Method for Query Optimization,* MS Thesis, Dept. of Computer Science, Univ. of Texas at Austin, Austin, TX.

[vEKo] van Emden, M.H., Kowalski, R., The semantics of Predicate Logic as a Programming Language, *JACM 23, 4,* 1976, pp. 733--742.

[Vie] Vieille, L. Recursive Axioms in Deductive Databases: the Query-Subquery Approach, *Proc. First Int. Conference on Expert Database Systems,* Charleston, S.C., 1986.

[WHD88] Warren, R., M. Hermenegildo, S. K. Debray, "On the Practicality of Global Flow Analysis of Logic Programs", Proc. of the 1988 Intl. Conf. and Symp. on Logic Programming, 1988.

[War] Warren, D.H.D., *An Abstract Prolog Instruction Set,* Tech. Note 309, AI Center, Computer Science and Technology Div., SRI, 1983.

[Z 85] Zaniolo, C., The representation and deductive retrieval of complex objects, *Proc. of 11th VLDB,* pp. 458-469, 1985.

[Zan1] Zaniolo, C., Prolog: a database query language for all seasons, in *Expert Database Systems, Proc. of the First Int. Workshop,* L. Kerschberg (ed.), Benjamin/Cummings, 1986.

[Z 86] Zaniolo, C., Safety and Compilation of Non-Recursive Horn Clauses, *Proc. First Int. Conf. on Expert Database Systems,* Charleston, S.C., 1986.

Two Problems in Recursive Query Optimization

R. Ramakrishnan
University of Wisconsin-Madison

May 1, 1989

Abstract

We identify two fundamental problems in processing recursive queries that have received little or no attention.

1 Query Processing: Nothing Ever Really Changes

Processing recursive queries is not so different from processing non-recursive queries: we must identify the space of logically equivalent queries, identify the space of access plans for each of these queries, and attempt to identify the (*query, access-plan*) pair that has the least associated run-time cost. Then, of course, we evaluate this query according to the chosen access plan.

This thesis is made explicit in [KRS88]. While the paper addressed the issue of detecting finiteness of answers (and intermediate results), it also provided a framework for query processing, consisting of the following steps:

1. Choose a sip — informally, the order in which body goals are to be solved — for each rule, for each (reachable) adornment.
2. Rewrite the program according to the set of chosen sips.
3. Evaluate the fixpoint of the rewritten program.

The important point, for our purposes here, is that this framework is essentially the same as that for non-recursive queries - we find a logically equivalent query (the rewritten program) and evaluate it. Thus, we omit special purpose evaluation algorithms, e.g. Prolog-style evaluation, from consideration. I think that this is a reasonable decision for database applications. (Note that we can still "mimic" those algorithms to the extent of not computing any facts that they do not compute, through rewriting according to the Magic Sets algorithm [BMSU86, BeR87, Ram88].)

2 Two Open Problems

There is a rich literature on program transformations that produce logically equivalent queries, e.g., [BMSU86, BeR87, NRSU89, RBK88, Sag87, SZ86]. [BaR86] provides a survey and further references. In contrast:

1. There is *no* literature on how to estimate the cost of a (query, access-plan) pair.
2. There is *no* literature on how to structure the space of such pairs in searching for a least-cost pair.

3 Discussion and Comments

I confess that I exaggerated a tad. For example, Lipton and Naughton have recently worked on estimating the size of the transitive closure of a relation [LN89]. A paper by Kathy Morris [Mor88] on re-ordering goals in rule bodies can be viewed as a step towards Problem (2). However, it is the case that these problems have not been addressed seriously, and no good solutions are in sight.

Problem (2) would be solved if someone wrote the equivalent of the Selinger et al. paper on access paths in System R [SACLP79] for recursive queries.

To address Problem (1), we must answer one central question:

- How do we estimate the *stage function* of a recursive query?

Informally, the stage is the number of iterations before the evaluation reaches a fixpoint, that is, the *depth* of the recursion.

Given the depth of a recursive query, I think conventional cost estimation techniques can be extended in a straightforward way to provide reasonable cost estimates for recursive queries that are evaluated using a fixed access plan in each iteration. But maybe I'm too optimistic.

One solution to estimating the depth is to maintain statistics from past queries. Does anyone have other ideas? (Note that a good parametrization of data is essential here. The model proposed in [BaR86] provides a rather limited first step.)

References

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Boston, Massachusetts, March 1986.

[BaR86] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16-53, Washington, D.C., 1986. Revised and reprinted in *Readings in AI and Databases*, Eds. M. Brodie and J. Mylopoulos, pages 376-430, 1988.

[BeR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[KRS88] Ravi Krishnamurthy, Raghu Ramakrishnan and Oded Shmueli. A framework for testing safety and effective computability in extended datalog. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 154-164, Chicago, Illinois, 1988.

[Mor88] Katherine Morris. Ordering conjuncts in datalog. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Austin, Texas, 1988.

[LN89] Richard Lipton and Jeffrey Naughton. Estimating the size of generalized transitive closures. To appear in *Proceedings of the International Conference on Very Large Databases*, 1989.

[NSRU89] Jeffrey F. Naughton, Yehoshua Sagiv, Raghu Ramakrishnan, and Jeffrey D. Ullman. Factoring can reduce arguments. To appear in *Proceedings of the International Conference on Very Large Databases*, 1989.

[Ram87] Raghu Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140-159, Seattle, Washington, August 1988.

[RBK88] Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. Optimizing existential datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–102, Austin, Texas, March 1988.

[Sag87] Yehoshua Sagiv. Optimizing datalog programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 349–362, Austin, TX, March 1987.

[SZ86] Domenico Sacca and Carlo Zaniolo. The generalized counting methods for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, 1986.

[SACLP79] Patricia Selinger, Mort Astrahan, Don Chamberlin, Raymond Lorie, and Ted Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23-34, Boston, Massacussetts, 1979.

Increasing the Flexibility of Query Optimization

Johann Christoph Freytag

European Computer-Industry Research Centre
Arabellastr. 17
D-8000 München 81, West Germany

jcf%ecrcvax.uucp@unido.EDU, ...!mcvax!unido!ecrcvax!jcf

Abstract

With extending current database technology to new application environments such as knowledge base management systems, the task of providing the right level of query processing and optimization for submitted user requests becomes more complex and time consuming than in a conventional database environment. Facing this increased complexity more **flexibility** in organizing query processing is necessary for adjusting this task to different needs. We therefore propose to organize query processing by **strategy rules**. Those should enable us to adjust the level of processing, including optimization, to different requirements resulting from internal (i.e. system) and and external (i.e. user) demands that both might vary over time.

1 Why more Flexibility?

With today's trend to extend currently existing (relational) database technology in various ways many of the existing concepts, techniques, and methods used need to be redefined and reexamined in the new context. In particular, the scope of query processing, in particular query optimization, must be broadened to guarantee an efficient evaluation of user requests that are expected to be more complex than requests in today's database management systems (DBMSs). The need to evaluate more demanding requests in extended database

to improve the evaluation of queries for a particular execution model. For example, in relational DBMSs physical optimization assumes an execution model whose basic operations such as selection, join and other algebraic operations, reflects the need for efficiently processing data stored on secondary storage. On the other hand, a deduction-based execution model (such as in Prolog), for example, requires other (physical) optimization techniques to speed up the evaluation of queries.

In contrast, we see methods of logical optimization as execution model independent. They should rewrite or transform a query without taking into account specific knowledge about "low level" operations for later evaluation. To give a more precise understanding of of the term logical optimization section 2 briefly mentions some of the existing techniques which are in general helpful for improving the evaluation of queries. Although much work has been done in this area only a few of those the logical optimization techniques have been implemented in current DBMSs only in very limited ways for various reasons.

When providing sophisticated optimization techniques the problem of how and when to apply those techniques becomes apparent. It is unfeasible and impractical to organize query processing and optimization statically, i.e. by a fixed sequence of techniques that are applied all the time. On the contrary, what is needed is an adaptable and flexible query processing scheme that can be adjusted to different queries depending on requirements specified by the user or derived from the request submitted.

We therefore suggest the use of **strategy rules** to express knowledge of how to organize the process depending on different parameters. In [Ull85] Ullman introduces strategy rules for the processing of recursive queries capturing the various alternatives that exist for alternative forms of the query. We apply the same idea in the general context of query processing to describe the organization of query processing in a flexible and adaptable manner. We discuss some initial ideas of strategy rules in section 3. Finally, we outline some general steps on how we intend to further develop the idea of a flexibly processing user requests in an extended database environment.

2 Techniques for Logical Optimization

In the past the development of query optimizer has mainly focused on implementing components for **physical** optimization. That is, most optimizers concentrate on immediately generating the "best" query evaluation plan from a given user query without considering other improvements, such as semantic query optimization [Kin81], first. We believe that the latter kind of optimization becomes more important, for example, in knowledge base systems where one expects more complex queries due to presence of rules and (possibly) a deduction component. For this reason we use the term **logical optimization** to emphasize the scope of optimization that is broader than semantic query optimization. Logical

optimization includes techniques such as[2]

- the integration of integrity constraints (e.g. semantic query optimization [SO87] and many others),
- the improvement of ranges in logic queries [Bry89],
- the standardization of queries achieving a canonical form (e.g. prenex normal form or miniscope form [Bry89]),
- the processing common subexpression (e.g. [Fin82] and others),
- the optimizing of multiple queries [Sel88],
- the rewriting of recursive queries (e.g. [BR86] and others),

This list is by no means complete. On the contrary, we expect this list of methods to grow in the future when current requirements will demand additional optimization techniques on the logical level.

3 Strategy Rules

To organize the techniques mentioned in the previous section we suggest the use of strategy rules. This concept of strategy rules – by no means new in the context of DBMSs – seems to proof advantageous whenever flexibility during the processing is required. Already Ullman organizes the translation of recursive queries around this concept [Ull85]. The motivation for the "event-condition-action" rules of the HiPAC project are a modular, execution independent specification of actions that are triggered if certain event happen and conditions are satisfied inside the DBMS [D+88].

We envisage "condition-action" rules as a way to specify the various processing steps in an flexible manner. The condition "tests" a given set of parameters and determines if this rule "applies". If so, the action part specifies how to process the given query completely or partially. In the latter case further rules are matched against the resulting query – possibly iteratively several times – until some "final form" is produced that is ready for execution.

The are various parameter that could determine the testing part of strategy rules such as:

syntactical properties of the query: How many relations/object types are accessed? Does the query contain views? Is it a recursive query? What is the syntactical

[2]The reader should note that the list of citations is not intended to be complete for each of the techniques mentioned. Rather the citations should be understood as "example references".

"complexity" (for example, for SQL queries the levels of nesting)? Does the query contain common subexpressions?

semantical properties of the query: Do there exist integrity constraints related to this query? If so is it likely that they help to improve the later evaluation? Should the query be rewritten into some canonical form that is beneficial for other methods of logical optimization?

statistical properties of the query: Are there small relations with only one or two tuples that justify partial evaluation before performing further evaluation? Does the query access relations with a large or only a few number of tuples? In the former case, a significant saving in evaluation time could amortize and justify an extensive optimization more easily than in the latter case.

dynamic properties: Is it advantageous to combine the evaluation of the query at hand with that of others (multiple query optimization)? Are there resource requirements or resource limits known in advance that should be considered? For example, should the query be optimized based on buffer space available at evaluation time?

user parameters: The user might request a cursory or extensive optimization depending on whether the query runs only once or several times, or whether it is part of a prototype or a production system. Based on the same kind of information the system could determine the extend of compilation of queries using techniques as outlined in [FG89] (i.e. to produce iterative programs), and could decide whether or not to keep the query evaluation plan resulting from the optimization process.

Of course, the success of strategy rules heavily depends on the efficient testing of those conditions and the accuracy that these tests provide. Both problems need further investigation. It also needs to be proved that we can formulate those strategy rules in an adequate manner such that they decide for the right kinds of optimization for the right queries at the right time. More practical experience is necessary to determine if the idea to organize logical optimization by strategy rules is feasible in practice.

4 Conclusion

In this position paper we briefly outlined how to organize logical optimization by strategy rules to provide the flexibility necessary for an extended processing of queries. We see a need for this kind of flexibility when extending current database technology to new environments such as knowledge base management systems. By emphasizing the structuring of the query optimization process we believe that we provide a framework that can be adapted to new methods and techniques developed in the future.

References

[BR86] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing. In *Proceedings ACM SIGMOD 1986, Washington, D.C.*, pages 16–52, May 1986.

[Bry89] F. Bry. Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. In *Proceedings ACM SIGMOD 1989, Portland, OR*, May 1989.

[D+88] U. Dayal et al. Rules are Objects Too: A Knowledge Model for an Active Object-Oriented Database System. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, Bad Ebernburg, September 1988.

[FG89] J.C. Freytag and N. Goodman. On the Translation of Relational Queries into Itarative Programs. *ACM Transactions on Database Systems*, 14(1), March 1989.

[Fin82] S. Finkelstein. Common Expression Analysis in Database Applications. In *Proceedings ACM SIGMOD 1982, Orlando, FL*, June 1982.

[Kin81] J. King. QUIST:A System for Semantic Query Optimization in Relational Databases. In *Proceedings VLDB 1981*, pages 510–517, August 1981.

[Sel88] T.K. Sellis. Multiple Query Optimization. *Transactions on Database Systems*, 13(1):23–52, May 1988.

[SO87] S.T. Shenoy and M. Ozsoyoglu. A System For Semantic Query Optimization. In *Proceedings ACM SIGMOD 1987, San Francisco, CA*, pages 181–195, May 1987.

[Ull85] J. D. Ullmann. Implementation of logical Query Languages for Databases. *ACM Trans. on Database Systems*, 10(3):289–321, Sept. 1985.

Heuristic-Based Semantic Query Optimization and Automatic Rule Derivation

Michael D. Siegel
Computer Science Department
Boston University
mds@bu-cs.bu.edu

1 Introduction

Semantic query optimization has been shown to be a useful method for reducing query processing costs [HA80, KI81a, KI81b, XU83, JA84, CH84, CH85, SH87, SI88a, SI88b]. Savings result from the use of rules, or integrity constraints, supplied by experts. Unfortunately, there has been little research targeted at the development of practical semantic query optimizers. In addition, a methodology for experts to establish a set of useful rules has not been developed. Thus semantic query optimization has not been used in numerous applications where it can contribute to more efficient database operation. In response, we have developed both a practical method for implementing semantic query optimizers and an automatic method for deriving rules for use in the optimization process.

2 Heuristic-Based Semantic Query Optimization

A key problem in developing a practical semantic query optimizer is deciding when and how to use a given integrity constraint. A constraint may not always be useful for optimization. Some constraints are effective only if indices are present, or when certain types of queries are asked. The standard method for identifying the most promising transformations is to use *transformation heuristics.*

The set of transformation heuristics determines the behavior of the optimizer. It is impractical for this set to be hard-coded into the optimizer; instead, the system administrator should be able to experiment with new heuristics, and to fine-tune old ones. Consequently, what is needed is a semantic query optimizer *generator.* That is, given a set of transformation heuristics, we should be able to produce an executable semantic query optimizer. Our optimizer generator is based on the conventional query optimizer generator [GR87a, GR87b] used in the EXODUS extensible database system [CA86]. It requires as input a

This work was funded by National Science Foundation grants DCR8407688, IST8408551 and IST8710137.

set of semantic transformations heuristics. The language used for specifying these transformations is derived from the EXODUS specification language for algebraic transformations. Unlike algebraic transformations, each semantic transformation must also specify routines for matching members of the rule set against proposed transformations [SI88b].

Because we designed our semantic query optimizer similarly to a conventional optimizer, we are able to compare them; we can examine how they duplicate certain functions, and see how they need to interact. We consider the advantages of a combined semantic/conventional optimizer over the traditional loosely-coupled approach. Such combined optimizers are generated by providing a rule-based specification that includes both conventional and semantic transformations.

An ostensible advantage of semantic query optimization is that the optimizer can answer some queries without accessing the database. For example, suppose an integrity constraint asserts that every employee is over 18. Then given the query asking for all employees of age 10, the optimizer can immediately detect that the answer must be empty. Although such behavior is correct here, this optimization has been erroneously extended to other types of queries. In our work we provide methods for the use of both tautologies and contradictions in finding solutions to queries without accessing the database.

Our research has examined the performance of generated semantic query optimizers. Using a sample database, performance statistics are obtained by submitting optimized and unoptimized queries to an implementation of the Wisconsin Storage System (WiSS) [CHO85].

3 Automatic Rule Derivation

The success of semantic query optimization depends on the existence of a set of useful rules. However, these rules are not necessarily those that would or could be specified by an expert. We have developed an automatic method for deriving a useful rule set. This method uses intermediate results from the optimization process to direct the search for learning new rules. Unlike user-specified rules, a system with an automatic capability can derive rules that may be true only in the current state of the database. An example of such a derived rule is "No employee makes more than $65,000." Although such rules may not always be true, it is useful for the system to take advantage of them as long as they are true. In addition, as the database is modified some derived rules may become invalid, while others become true. Some derived rules may lose their usefulness. A system that maintains a set of derived rules is flexible in a time-varying world.

The process of automatic rule derivation is known in the field of Artificial Intelligence as inductive or heuristic learning [BLU82, DA82, LE83a, LE83b, LI80, MI83, WA70]. Given a domain such as medical diagnosis [BLU82], a learning system derives appropriate rules and constraints by examining data or sequences of operations on the data. Usually, heuristics are used to limit the search for effective rules. They also determine the effectiveness of the system. In a similar manner we can use heuristics to identify the characteristics of the rules to be derived and to search the database for these rules. It seems promising and natural to apply the techniques of inductive learning to rule derivation for database systems.

The derivation of rules from the database, a difficult search problem, is accomplished

using information obtained from the database usage pattern, a set of query transformation heuristics, and query cost formulas. The use of a query history for database optimization has been examined in other self-adaptive database optimization techniques such as index selection [CHA76], view materialization or the use of temporaries [FI82, JA85, SE86], and physical database restructuring [AH84, NA84]. In automatic rule derivation, the query history assessment is important in determining the characteristics of rules that are useful in semantic query optimization.

Once rules have been derived from the database, their value in semantic query optimization must be monitored. In particular, the size and quality of the rule set must be controlled. In addition, the rule set must remain valid in the presence of updates. Unlike integrity constraints, not all derived rules remain valid in all database states. Thus violated rules must be deleted or modified to reflect the new database state.

Along with developing methods for implementing semantic query optimization, our research has defined methods for identifying, deriving and maintaining rules for use in semantic query optimization. The automatic rule derivation process works in parallel with the semantic query optimizer, sharing some knowledge sources and using some intermediate results from the optimizer. These methods can be used to derive rules for query optimization in conjunction with or in lieu of a human expert. The savings that result from this method will accrue when rules that are derived can be used repeatedly in the optimization process. The success of semantic query optimization may depend on the ability of this automated process to maintain a set of useful rules in a changing database environment.

4 Conclusions and Future Research

In summary, we have developed, based on the conventional optimizer generator described in [GR87a], a data-model independent semantic query optimizer generator. This generator makes it possible for a database implementor to provide a specification of semantic transformations and to generate from these specifications an executable optimizer. Using an optimizer based on the relational model and a sample database, we have been able to show that the savings expected from this type of optimization are real. Finally, we have described a method for automatically deriving rules for use in optimization. This process of rule derivation reduces the dependency on human experts and provides a means for the optimization process to follow changes in the database usage pattern, making semantic query optimization possible and effective in applications that have limited access to expert knowledge.

Our research describes advances in both database and artificial intelligence research in a time when these two fields are becoming increasingly interdependent, but there exists a number of areas for further research.

First, we would like to consider the extension of semantic query optimization and automatic rule derivation to semantic data models. All of the research to date has focused on the relational model. However, the added structure of semantic models appears to be better suited to semantic transformations, and useful integrity constraints seem to be much more prevalent. Some preliminary work in this area is discussed in [SI88a, SI88b].

Next, we would like to consider more general integrity constraints. Although simple rules

are common, there are other useful classes of constraints, such as functional dependencies and inclusion dependencies [JA84]. We need to examine how easily our optimizer generator can be adapted to handle these constraints. It is also possible to consider the derivation of these more complex rules but this may prove difficult without some assistance from a domain expert.

We would like to examine more closely the various methods for improving the performance of both the semantic query optimizer and the rule derivation process. These include methods for limiting the application of heuristics. The use of phases [GR87b] and heuristic performance statistics [HA80] provide some added control. Additional system-derived or user-specified data may be useful in fine tuning the optimization process [SI88b]. But the effect of the interaction of these techniques on the optimization process is not well understood. In addition, further experimentation is needed to study the interaction between algebraic and semantic transformations.

Heuristic-based semantic query optimization is dependent on the ability of the database implementor to supply a set of useful semantic transformation heuristics. But the database implementor has little or no guidance in determining if he or she has written a complete and consistent set of heuristics. Methods are needed to assist the implementor in developing heuristics that fit these requirements.

Additionally, there are numerous questions that remain about the derivation process. First we need to determine the savings that can be realized from semantic query optimization using automatic rule derivation. In particular, when is it likely that we will generate meaningful rules? What types of rules are most likely to be derivable? What types of data models are appropriate for semantic query optimization using automatic rule derivation? What types of applications are appropriate for these methods? We hope to address these questions and others using the present implementation.

Once rules have been derived from the database, they must be maintained in the presence of updates. Several methods for maintaining a valid set of derived rules were described in [SI88b]. These methods include explicit reference to exceptional instances and rewriting of rules to account for violations. It is a matter of future research to implement these and other methods in order to discover which are cost effective in a given application. Managing derived rules in the presence of updates poses many problems for both artificial intelligence and database research. For example, the ability of a system to learn from exceptions and the explicit use of exceptions in rule maintenance and other database operations.

In summary, this paper describes a new approach to query optimization by first establishing methods for heuristic-based semantic query optimization, and then by defining automatic rule derivation as a learning process that can be used to supply the optimizer with a set of useful rules. The success of semantic query optimization in reducing query processing costs will depend on learning methods like this to provide the database with a useful set of rules in a changing database environment.

5 References

[AH84] AHAD, RAFIUL (1984), User-Assisted Design and Evolution of Physical Databases, Ph.D. Thesis, University of Southern California.

[BLU82] BLUM, ROBERT (1982), Discovery, Confirmation, and Incorporation of Causal Relationships from a Large Time-Oriented Clinical Data Base: The RX Project, *Computers and Biomedical Research*, Volume 15, pp. 164-187.

[CA86] CAREY, MICHAEL, et. al. (1986), The Architecture of the EXODUS Extensible DBMS: A Preliminary Report, *Proceedings of the International Workshop on Object-Oriented Database Systems*, pp. 52-65.

[CH84] CHAKRAVARTHY, U., D. FISHMAN, AND J. MINKER (1984), Semantic Query Optimization in Expert Systems and Database Systems, *Proceedings of the First International Conference on Expert Database Systems*, South Carolina, pp. 326-340.

[CH85] CHAKRAVARTHY, UPEN (1985), Semantic Query Optimization in Deductive Databases, Ph.D. Thesis, University of Maryland.

[CHA76] CHAN, ARVOLA (1976), Index Selection in a Self-Adaptive Relational Data Base Management System, Masters Thesis, Massachusetts Inst. of Technology, TR-166.

[CHO85] CHOU, H-T., DAVID DEWITT, RANDY KATZ, and ANTHONY KLUG (1985), Design and Implementation of the Wisconsin Storage System, Software Practice and Experience, Vol. 15(10), pp 943-962.

[DA82] DAVIS, RANDALL AND LENAT, DOUGLAS (1982), *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill Advanced Computer Science Series.

[FI82] FINKELSTEIN S. (1982), Common Expression Analysis in Database Applications, *Proceeding of the 1982 ACM-SIGMOD Conference on Management of Data*, Orlando, Fla., pp. 235-245.

[GR87a] GRAEFE, GOETZ and DAVID DEWITT (1987), The EXODUS Optimizer Generator, *Proceeding of the 1987 ACM-SIGMOD Conference on Management of Data*, San Francisco, CA, pp. 160-171.

[GR87b] GRAEFE, GOETZ (1987), Rule-Based Query Optimization in Extensible Database Systems, Ph.D. Thesis, University of Wisconsin.

[HA78] HAMMER, MICHAEL and SUNIL SARIN (1978), Efficient Monitoring of Database Assertions, *ACM-SIGMOD International Conference on Management of Data*, Austin, Texas., pp. 38-49.

[HA80] HAMMER, MICHAEL and STANLEY B. ZDONDIK (1980), Knowledge-Based Query Processing, *Proceedings 6th VLDB Conference*, Montreal, pp. 137-146.

[JA84] JARKE, M. (1984), Semantic Query Optimization in Expert Systems and Database Systems, *Proceedings of the First International Conference on Expert Database Systems*, South Carolina, pp. 467-482.

[JA85] JARKE, M. (1985), Common Subexpressions Isolation in Multiple Query Optimization, In [KIM85], pp. 191-205.

[KI81a] KING, JONATHAN J. (1981), QUIST: A System for Semantic Query Optimization in Relational Databases, *Proceedings 7th VLDB Conference*, Cannes, pp. 510-517.

[KI81b] KING, JONATHAN J. (1981), Query Optimization Through Semantic Reasoning, Ph.D. Thesis, Stanford University.

[LE83a] LENAT, DOUGLAS (1983), Theory Formulation by Heuristic Search. The Nature of Heuristics II: Background and Examples, *Artificial Intelligence*, Volume 21, Number 1 and 2, pp. 31-60.

[LE83b] LENAT, DOUGLAS (1983), EURISKO: A Program that Learns New Heuristics and Domain Concepts. The Nature of Heuristics III: Program Design and Results, *Artificial Intelligence*, Volume 21 , Numbers 1 and 2, pp. 61-98.

[LI80] LINDSAY, R., B. BUCHANAN, E. FEIGENBAUM, and J. LEDERBERG (1980), *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project*, McGraw-Hill.

[MI83] MICHALSKI, R., J. CARBONELL and T. MITCHELL (1983), *Machine Learning*, Tioga Publishing.

[NA84] NAVATHE, SHAM, et. al. (1984), Vertical Partitioning Algorithms for Database Design, *ACM Transactions on Database Systems*, December, pp. 680-710.

[SE86] SELLIS, TIMOS (1986), Global Query Optimization, *Proceedings of the 1986 ACM-SIGMOD International Conference on the Management of Data*, Washington D.C., pp. 191-205.

[SI88a] SIEGEL, MICHAEL (1988), Automatic Rule Derivation for Semantic Query Optimization, *Proceedings from the Second international Conference on Expert Database Systems*, Tysons Corners, Virgina, pp. 371-385.

[SI88b] SIEGEL, MICHAEL (1988), Automatic Rule Derivation for Semantic Query Optimization, Ph.D. Thesis, Boston University.

[SH87] SHENOY, SREEKUMAR and MERAL OZSOYOGLU (1987), A System for Semantic Query Optimization, *Proceedings of the 1987 ACM-SIGMOD International Conference on the Management of Data*, pp. 181-195.

[WA70] WATERMAN, D. (1970), Generalization Learning Techniques for Automating the Learning of Heuristics, *Artificial Intelligence*, Volume 1, pp. 27-120.

[XU83] XU, DING (1983), Search Control in Semantic Query Optimization, University of Massachusetts, Department of Computer Science, Tech Report TR83-09.

Semantic Query Optimization in Distributed Databases A Knowledge-Based Approach

H.J.A. van Kuijk * *P.M.G. Apers*

University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

1 Introduction

Conventional query optimization is based principally on syntactic considerations. To facilitate the optimization process, certain characteristics inherent to the application being modeled could be used [Hammer 80,King 81]. To be applicable, this knowledge should be expressed in a formalism suitable to the query optimizer.

Our approach described in this extended abstract provides a framework to solve a number of existing and new problems encountered during (semantic) query optimization in a (distributed) database system: representation and application of semantic knowledge, deriving more efficient schedules for selection and join operations, defining and using horizontal fragmentation knowledge, estimating more accurately the profiles of intermediate results. Our framework is based on a hierarchy of constraints to explicitly represent application knowledge to be applied to arrive at more efficient query evaluation plans.

The rest of this short paper is organized as follows. In section 2, a hierarchy of static constraints is introduced in the context of representing application knowledge to be used by a query optimizer. In section 3, our framework is demonstrated to unify the solution to a number of problems to be solved during query optimization. In section 4, our knowledge-based approach is introduced. Finally, in section 5 some conclusions are given.

2 Constraints

The relational data model has its limitations concerning capturing useful properties of data such as additional restrictions on data values and specifications of how the data may be related. In our research, a database is defined by a database scheme (DBS). The DBS consists of a finite set of relation schemes $\{RS\}$, and a finite set of constraints $\{C\}$.

$$DBS = < \{RS\}, \{C\} > .$$

Constraints provide a means of explicitly representing certain characteristics inherent to the application being modeled. They should be expressed as well-formed formulas of a formalism such as predicate calculus that is amenable to exact specification and verification. The set of constraints $\{C\}$ should be satisfiable.

*E-mail: utrcu1!vankuyk@mcvax.uucp

In general, two different classes of constraints are distinguished: dynamic constraints and static constraints. Dynamic constraints are operation-oriented conditions a database transition must satisfy. Static constraints are conditions a database state must satisfy.

Static constraints can be used to represent explicitly conventional dependencies (referential constraints, key and functional dependencies, multi-valued and join dependencies) and semantic constraints (value constraints, implication constraints, subset constraints). The following constraint hierarchy is appropriate: domain constraints, attribute constraints, tuple constraints, relation constraints, and database constraints. In this short paper, only the first three constraints are considered.

Definition 2.1 A domain D_A underlying an attribute A is defined as a subset of an underlying base domain (BD) by a one-place predicate formula $DC(v)$ representing a *domain constraint*: $D_A = \{v \mid v \in BD \wedge DC(v)\}$.

A domain constraint DC can be used to explicitly define a domain D_A of an attribute A as a subset of some underlying base domain (BD).

Definition 2.2 An *attribute constraint* on an attribute A is a one-place predicate formula defining a subset of values V of the domain D_A underlying attribute A: $V = \{v \mid v \in D_A \wedge AC(v)\}$.

An attribute constraint AC can be used to explicitly represent certain characteristics inherent to an attribute. For the set of constraints $\{C\}$ to be consistent, an attribute constraint $AC(a)$ must always imply the domain constraint $DC(a)$, denoted $AC(a) \rightarrow DC(a)$.

Definition 2.3 A *tuple constraint* on n attributes is an n-place predicate formula defining a subset S of the cartesian product of the n domains underlying the attributes referenced by the predicate formula: $S = \{t \mid t \in D_1 \times D_2 \times \cdots \times D_n \wedge TC(t)\}$.

A tuple constraint TC is an inter-attribute constraint explicitly representing certain useful characteristics concerning combinations of attribute values. In section 3, the constraints defined in this section are applied to solve a number of optimisation problems.

3 Query Optimization

In this section, the hierarchy of constraints introduced in the previous section is demonstrated to be a uniform framework to solve a number of problems encountered during query optimisation.

3.1 Semantic Query Optimization

Because query optimisation is known to be NP-hard, a heuristic approach seems justified. Weak search methods, such as the A^*-algorithm, cannot be applied because of the lack of a well-defined goal state[1].

Our approach is to divide the overall problem into several smaller stages. Sometimes some local, but not global, tests of optimality are possible for the individual stages. A consequence of this approach is a solution that might not be optimal but is considered *satisfying*. The following stages are distinguished.

Query preprocessing : the query is transformed into a canonical internal expression.

Query expansion : knowledge of the application is used to expand selection and join predicates.

Query reduction : all semantically redundant selection and join predicate formulas are removed.

Query postprocessing : the resulting query expression is transformed further into a schedule.

[1] In general, it is impossible to detect whether a solution is *optimal*.

Selection	:	$TC_{\sigma_F(R)}$	=	$TC_R \wedge F$.
Projection	:	$TC_{\pi_A(R)}$	=	$TC_R[A]$.
Union	:	$TC_{R \cup S}$	=	$TC_R \vee TC_S$.
Intersection	:	$TC_{R \cap S}$	=	$TC_R \wedge TC_S$.
Difference	:	TC_{R-S}	=	TC_R.
Cartesian Product	:	$TC_{R \times S}$	=	$TC_R \wedge TC_S$.
Join	:	$TC_{R \bowtie_F S}$	=	$TC_R \wedge TC_S \wedge F$.

Table 1: Tuple constraints resulting from relational operations.

Our framework of constraints provides a mechanism for representing semantic knowledge to be used during the expansion and reduction stages. Consider the following *if-then* rules defining relationships between intervals I referencing the attributes a and b:

$$\begin{array}{lcl} rule_1 & : & \textbf{if } I_{a_1} \textbf{ then } I_{b_1} \quad R_1(a,b) = (I_{a_1} \rightarrow I_{b_1}) = (\neg I_{a_1} \vee I_{b_1}). \\ rule_2 & : & \textbf{if } I_{a_2} \textbf{ then } I_{b_2} \quad R_2(a,b) = (I_{a_2} \rightarrow I_{b_2}) = (\neg I_{a_2} \vee I_{b_2}). \end{array}$$

The tuple constraint resulting from both the *if-then* rules is constructed by having the conjunction of $R_1(a,b)$, $R_2(a,b)$, and the available attribute constraints $AC(a)$ and $AC(b)$:

$$\begin{array}{rcl} TC(a,b) & = & R_1(a,b) \wedge R_2(a,b) \wedge AC(a) \wedge AC(b). \\ & = & (\neg I_{a_1} \vee I_{b_1}) \wedge (\neg I_{a_2} \vee I_{b_2}) \wedge AC(a) \wedge AC(b). \end{array}$$

In our approach, it is possible to apply knowledge in the form of constraints (tuple constraints, attribute constraints, and domain constraints) during the process of (semantic) query optimisation. A more detailed description of the techniques discussed here is given in [Kuijk 88].

3.2 Selection and Join Operations

In our approach to query optimisation, selection and join predicates play a central role. Knowledge about the domain of application represented as constraints can be applied to augment selection and join predicates resulting in more efficient query evaluation plans.

In case of a selection operation $\sigma_F(R)$, all the tuples of operand relation R are defined to satisfy the tuple constraint TC_R on R. Without influencing the semantics of the operation, the selection predicate can be expanded to $F \wedge TC_R$. Given this expanded selection predicate, the following interesting situations are distinguished.

1. If $F \wedge TC_R$ is unsatisfiable, then *none* of the tuples of R can ever satisfy the selection predicate F. The entire operation should be discarded.

2. If $TC_R \rightarrow F$, then *all* the tuples of R must satisfy the selection predicate F. The query expression should be adjusted accordingly since $\sigma_F(R) = R$.

In our approach, the optimiser can detect the above situations. Unnecessary operations can therefore be avoided. A similar approach is applied to detect and discard unnecessary join operations. Constraints can thus be applied to generate more efficient schedules. In general, the operand relations of selection and join operations can result from previous operations. Table 1 shows how to obtain the tuple constraints on the relations resulting from relational operations [Kuijk 88][2].

[2]This resembles the notion of *qualified* relations described in [Ceri 84].

3.3 Horizontal Fragmentation

Our framework of constraints provides an elegant means to define the partitioning of global relations into horisontal fragments. In our approach, a horisontal fragmentation is defined by an attribute constraint (single fragmentation attribute) or, in general, a tuple constraint (more than one fragmentation attribute).

A horisontal fragmentation should satisfy the reconstruction, completeness, and disjointness conditions [Ceri 84]. In our framework of constraints, it is possible to check whether these conditions are satisfied (for example during fragmentation design).

Being expressed explicitly as constraints, it is possible for the query optimiser to apply this fragmentation knowledge during the process of generating more efficient schedules. Consider the following query:

$$\begin{aligned} Q &= \sigma_F(R_g). \\ &= \sigma_F(R_1 \cup R_2 \cup \cdots \cup R_n). \\ &= \sigma_F(R_1) \cup \sigma_F(R_2) \cup \cdots \cup \sigma_F(R_n). \\ &= \sigma_{F \wedge TC_{R_1}}(R_1) \cup \sigma_{F \wedge TC_{R_2}}(R_2) \cup \cdots \cup \sigma_{F \wedge TC_{R_n}}(R_n). \end{aligned}$$

Extension of the selection predicates enables the query optimiser to apply the techniques discussed before to avoid unnecessary operations during query evaluation.

3.4 Estimating Intermediate Results

The selection among alternative schedules is guided by their expected performance according to some cost model. The optimiser needs a provision to estimate the profiles of relations resulting from relational operations [Ceri 84].

Estimating the cardinality of intermediate results can be translated to calculating the size of the n-dimensional bodies defined by the n-place constraint predicate formulas on the result relations. In [Kuijk 88], a *constraint normal form* (CsNF) is defined. Informally, a constraint predicate formula is in CsNF if it is a predicate formula in conjunctive normal form with mutually disjoint conjuncts. The property of mutually disjoint conjuncts is the key to more accurate estimations. For example, in case of a union, the number of tuples in the intersection can be determined more accurately and are counted only once[3]. Depending on the problems, in [Kuijk 88] our approach is demonstrated to result in considerably more efficient estimates (in case of selection operations, improvements of 50 % are achievable).

4 A Knowledge-Based Approach

A *knowledge system* is a computer program with a clear separation between domain-specific knowledge and domain-independent inference knowledge. Knowledge to be used during query optimisation should be made explicit. An explicit representation enables the optimisation knowledge to be managed (added, deleted, modified). In our approach, the following knowledge sources are distinguished:

Strategy Knowledge : this knowledge reflects the strategies to attack the problem of query optimisation (knowledge about available optimisation strategies and heuristics to select among them).

Domain Knowledge : the system needs knowledge about the various subprocesses of query optimisation (including syntactic and semantic transformation techniques, estimating intermediate results, determining efficient join sequences, etc.).

Application Knowledge : the system needs knowledge about the contents of the database (database schemes, relation schemes, definitions of domains and attributes, statistics, constraints, etc.).

[3] $card(R \cup S) = card(R) + card(S) - card(R \cap S)$.

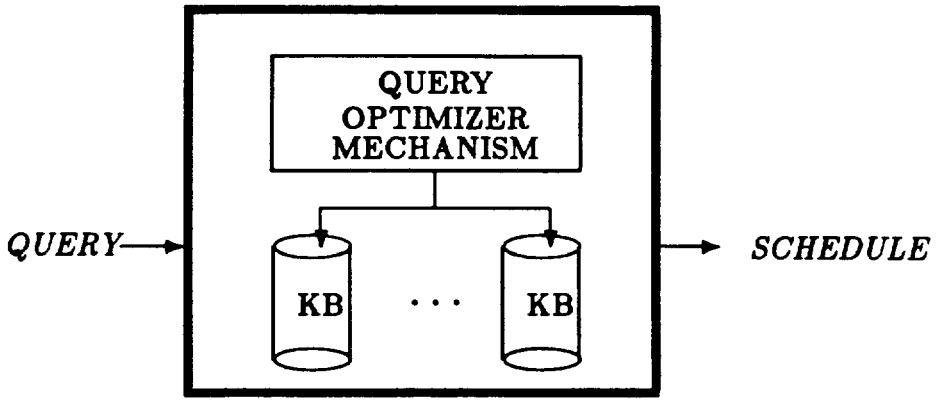


Figure 1: Architecture of the knowledge-based query optimiser.

Implementation Knowledge : the system needs knowledge representing the characteristics of system-dependent parts such as available algorithms for the (relational) operators, storage structures and access mechanisms, hardware characteristics (CPU, disks, network).

The architecture of our knowledge-based query optimiser is given in figure 1. Distinguishing different stages during the process of query optimisation allows the knowledge base of the system to partitioned into well-defined smaller modules. Instead of searching a potentially large and growing knowledge base, the inference mechanism can focus on one of the smaller modules during the various stages of the optimisation process. The power of a knowledge-based approach is its flexibility. It facilitates an easy extension to new optimisation knowledge and other environments (extensible databases, complex objects, new operators, new storage structures and access mechanisms).

5 Conclusions

In our approach to (semantic) query optimisation in a distributed database, the explicit representation of potential query optimisation knowledge inherent to the application being modeled is a central issue. A hierarchy of static constraints is demonstrated to be the foundation of a framework unifying a number of problems encountered during query optimisation: the representation and application of semantic knowledge, the manipulation of selection and join predicates, the definition of a horizontal partitioning of global relations, the estimation of profiles of intermediate results.

Our query optimiser is implemented as a knowledge system. Since we decompose the overall optimisation process into a number of smaller independent stages, the potentially large knowledge base of the optimiser is partitioned into a number of well-defined modules. A number of optimisation knowledge sources are distinguished: strategy knowledge, domain-dependent knowledge, application-dependent knowledge, implementation knowledge. The power of our system is its flexibility. It allows an easy extension to new environments (extensible databases, complex objects, new operators, new access and storage mechanisms, etc.).

References

[Apers 83] P.M.G. Apers, A.R. Hevner, and S.B. Yao, Optimisation Algorithms for Distributed Queries, *IEEE Transactions on Software Engineering*, Volume SE-9, Number 1, pp. 57-68, January, (1983).

[Brock 80] E.O. de Brock, *Tables, Table Variables, and Static Integrity Constraints*, Memorandum 80-12, Eindhoven University of Technology, (1980).

[Ceri 83] S. Ceri and G. Pelagatti, Correctness of Query Execution Strategies in Distributed Databases, *ACM Transactions on Database Systems*, Volume 8, No. 4, pp. 577-607, December, (1983).

[Ceri 84] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, (1984).

[Chang 73] C.L. Chang and R.C.T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, Orlando, Florida, (1973).

[Date 83] C.J. Date, *An Introduction to Database Systems*, Volume II, Addison-Wesley, Reading, Massachusetts, (1983).

[Date 86] C.J. Date, *Database Systems*, Volume I, Fourth Edition, Addison-Wesley, Reading, Massachusetts, (1986).

[Genes 87] M.R. Genesereth and N.J. Nilsson, *Logic Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers, Los Altos, California, (1987).

[Hammer 80] M. Hammer and S.B. Zdonik, Knowledge-Based Query Processing, *Proceedings of the* 6^{th} *Conference on Very Large Databases*, (1980).

[Jack 86] P. Jackson, *Introduction to Expert Systems*, Addison-Wesley, Reading, Massachusetts, (1986).

[Jarke 84] M. Jarke and J. Koch, Query Optimisation in Database Systems, *ACM Computing Surveys*, Volume 16, No. 2, June, (1984).

[Jarke 85] M. Jarke, J. Koch, and J.W. Schmidt, Introduction to Query Processing, *Query Processing in Database Systems*, Eds. W. Kim, D.S. Reiner, and D.S. Batory, Springer, New York, pp. 3-28, (1985).

[King 81] J.J.King, QUIST: A System for Semantic Query Optimisation in Relational Databases, *Proceedings of the* 7^{th} *Conference on Very Large Databases*, (1981).

[Korth 86] H.F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, New York, (1986).

[Kuijk 88] H.J.A. van Kuijk, *The Application of Constraints in Query Optimization*, University of Twente, Computer Science Department, Report INF-88-55, December 1988.

[Kuijk 89] H.J.A. van Kuijk, *The Application of Dependencies in Query Optimization*, University of Twente, Computer Science Department, Report INF 89-xx, (in preparation).

[Maier 83] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, (1983).

[Seling 79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, Access Path Selection in a Relational Database System, *Proceedings of the ACM Sigmod International Conference on the Management of Data*, pp. 23-34, (1979).

[Simm 84] M.K. Simmons, Artificial Intelligence for Engineering Design, *Computer-Aided Engineering Journal*, April, (1984).

[Ullman 82] J.D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, Maryland, (1982).

Intelligent query answering in Expert Database Systems using a heuristic approach

Suk-chung Yoon and Lawrence J. Henschen
Dept. of Electrical Engineering and Computer Science
Technological Institute
Northwestern University
Evanston, IL 60208

EXTENDED ABSTRACT

In recent years, there has been an increasing awareness that such seemingly disparate fields such as databse and artificial intelligence are related in many ways. Expert systems in artificial intelligence and database management systems are two technologies that having evoloved along different lines, but are now beginning to merge together toward the common goal of knowledge management systems. The merging of these two technologies results in the emergence of a new technology, namely expert database systems. The resulting expert databse system will benifit from the deductive problem solving capability of an expert system as well as the sophisticated and efficient management of a large database of facts and the enforcement of data reliability, integrity and security in a database management system. Other important advantages of the merging these technologies include enhanced query languages, semantic query processing and optimization, intelligent interfaces, etc.

Database systems have been designed to effectively find all of the feasible solutions which satisfy such a query rather than a few good solutions. But expert systems traditionally have been used to find only one or few good solutions for a query rather than all feasible solutions. A good example is the air-line reservation system where a passenger is not interested in looking at all of the feasible routes between two cities: instead he/she is interested in the optimal route or a set of "good" routes. A system may not want to confuse the user with all possible flights but rather choose a few good looking ones. Our objective is to purpose a general strategy for the design and implementa-

tion of an intelligent general-purpose expert database system that includes findfew and findall queries. Our study uses the semantic knowledge(integrity constraint) to limit the search space and also heuristics(domain-specific knowledge, expert knowledge, approximate knowledge etc.) as search constraints to order the search space to explore and improve the quality of answers in the case of a findfew query. This paper also addresses the efficient query evaluation and optimization based on the heuristic methods. In this paper, we use the first order logic for expressing the rules, organizing the knowledge and for reasoning; it gives much greater expressive power, conciseness and flexibility. Our expert database system is composed of four parts: (i) an extensional database(EDB), which consists of facts, represented by clauses with only one positive literal containing no variable, (ii) an intensional database(IDB), which consists of deductive rules, represented by non-recursive and function-free horn clauses, (iii) a set of integrity constraints(IC), which specify semantic relationships expressed only over the EDB relations, (iv) a set of heuristics.

We divide the process of intelligent query answering into two phases, the pre-processing phase and the query processing phase. In the pre-processing phase, there are four parts as follows: pre-processing of the IDB, pre-processing of the ICs, pre-processing of the heuristics and finding relevent integrity constraints and heuristics for each predicates. There are two parts for the pre-processing of the IDB. One is the rule factorization and the other is rule compilation. In the rule factorization, if two of IDB rules that have same head literal share a common initial segment, we merge these rules and factorize the initial segment. There are some advantages for rule factorization. We will avoid repeating the same operation at run time and also reduces storage because we don't need separate storage for the common initial parts of rules. In the rule compiler, a graph model is developed to facilitate the search through the rule set of an expert database system. That is, the rules in the IDB are transformed into a graph structure to process efficiently.

Definition: Suppose we are given IDB rules. Following is the definition of rule-graph G=(V,E).

(i) V={ set of node }. There are two kinds of nodes in the graph. C-node is a node for the consequent part in a IDB rule. A-node is a node for the antecedent part in a IDB rule.
(ii) E={(x, y) | x, y are in V }. There are two kinds of edges in the graph. For each IDB rule, there is an edge from C-node to A-node. This edge is called J-edge(it means that we derive the consequent part by the join of literals in the antecedent part). For each A-node, if A-node includes IDB literal which is unifiable, then there is an edge from the IDB literal in the A-node to C-node of the IDB literal. This edge is called U-edge(It means unifiable edge).
We attach unifiers on U-edges in the rule compile-time for reducing the run time. The rule-graph is very useful in the situation in which the rule update is occured frequently because we just modify the corresponding node in the rule graph. Precompiling rules into the graph facilitates the maintenance and management of the IDB and improve the performance. To complete the pre-processing of integrity constraints, we need two parts. First, derive all possible new integrity constraints derivable from the original once by resolution. Second, convert all integrity constraints to the variable substituted form: this will allow more effective use of the ICs. We use implicity constraints that are useful in searching the answer. *Implicity constraint are not explicitly supplied* but are generated during searching the solution of a specific set of goals. In pre-precessing the heuristics, domain specific knowledge, expert knowledge and approximate knowledge can be coded into rules using first order predicate logic to assist in problem solving. After finishing pre-processing of the IDB, IC and heuristics, we need to relate the relevent integrity constraints and heuristics to the IDB rules. That is, integrity constraints and heuristics are combined into the graph. We propose an algorithm using the modified partial subsumption technique. Restriction implied by relevent integrity constraints and heuristics are collected and used in optimizing the queries.

In the query processing phase, upon receiving a query, first the system will extract the relevant portion from the graph. Second the system will transform the given query with the integrity constraints and heuristics into a form that can be evaluated more

efficiently than the original query. For example, we may eliminate the unnecessary and redundant restrictions, find the transitive constraint, introduce a constraint for efficient searching and move a constraint across the join boundary, etc. In this step, we suggest some heuristics that choose the profitible ones among the integrity constraints and heuristics in the query context. After the transformation process, we have a semantically equivalent query. Third, the system receives the transformed query and generates the optimized processing tree. The hierarchical organization of the tree represents the order in which the nodes in the tree are to be evaluated. We propose a heuristic method which decides the order of the predicate evaluation using a corresponding attribute matching graph. The basis for the ordering of subgoals in the body of a clause is sharing of variables. For example, we assume that P1(x,y), P2(x,w) P3(y,t,z) and P4(y,z) are the extracted EDB relations. If the value of x is known, the coresponding attribute matching graph is

In this case, we convert a cycle into a tree by finding the minimum spanning tree using join selectivity. The above three processes are combined into a part of the system called the extractor. Fourth, the evaluator receives the optimized processing tree from the extractor and converts the optimized tree into an algebraic equation. We propose an algorithm to convert optimized tree to algebraic equation. If there is more than one path in the tree for a findfew query, we send the preferred path first. If the preference of the rule is known, we can evaluate first the rule that has higher preference in the generated processing tree. The preference is defined so that using the first rule generates more useful answers than subsequent ones. If there is no preferred path in the tree for a findfew query, we suggest heuristic functions to decide the order of paths in the tree. For example, the cost function of each path, the expected intermediate results, etc. In the evaluator step, we can elimiate the need to reevaluate the lead-

ing common subexpression in the optimized tree and thus, eliminate a large amount of redundant processing. Fifth, the search controller receives the algebraic equation from the evaluator and executes the relational algebra equation. In this step, if a given query is findfew, then selective search can also be performed on the portion of the EDB relations based on different criteria, such as the importance of the data, user preference and closeness and easiness in accessing data.

In conclusion, our approach develops efficient strategies and algorithms for supporting not only findall queries but also findfew queries where only a few intelligent solutions are needed by using the integrity constraints, domain knowledge and some heuristic methods in some steps to improve the query processing time.

An Example of Semantic Query Optimization as a Technique for Improving Query Performance in Federated Systems

Sandra Heiler

Xerox Advanced Information Technology
4 Cambridge Center
Cambridge, MA 02139
heiler@xait.xerox.com

Abstract

In federated systems, optimizations that are based on characteristics of the components of the federation are extremely difficult. We focus on optimizations that are based on object structure that can be captured by a global schema for the federation. This report briefly describes an example of such semantic query optimization using simple structures called "inclusion graphs". They capture set inclusion relationships that can be exploited in processing certain kinds of queries. Inclusion graphs can be used in federated systems regardless of the characteristics of components that provide the implementations of storage and manipulations of objects.

1. Introduction

Federated systems may provide the only safe and practical approach to configuring large, complex computing environments that must incorporate a legacy of existing systems. In a federation, autonomous components are loosely coupled to allow them to interchange information and provide services to one another, without requiring changes to the underlying software or data. Federation also provides a natural path for extending existing environments with new technologies. Systems can be added to the federation one-at-a-time without risking the safety of the existing operation.

An object-oriented approach is a natural way both to model and to implement these systems. An object model provides a uniform interface to data and operations, which are implemented by the underlying heterogeneous components of the federation. An Object Management System (OMS) provides services to map requests given in terms of the interface into invocations of the operations and data of the underlying software tools, DBMSs, file systems and databases.

We are concerned with providing techniques for improving query performance in federated systems constructed from heterogeneous underlying data servers. Optimizations that are based on characteristics of the implementations are extremely difficult in such systems. Few formalisms are available for describing the behavior of the underlying systems to allow the optimizer to generate requests that avoid their particular weaknesses or exploit their particular strengths.

The focus of our work is on strategies for optimization in federated systems that are (nearly) independent of the particular characteristics of the underlying implementations. We are especially interested in techniques for optimization based on object *structure*. Below, we give an example of how ancillary structures that capture set inclusion semantics can be used to improve query performance in a federated system that is implemented over heterogeneous components, regardless of the characteristics of the underlying systems.

Two streams of prior work are applicable to optimization for federated systems: global optimizations in systems that integrate heterogeneous databases (or "multi-database" systems), as described in [LaRo82], [LiAb86] and [Nava86], and semantic query optimization, as described in [HaZd80] and [Chak88]. Our interest is in extending some of these concepts to exploit structural information and other metadata that can be captured in an object model to improve query performance.

2. Context

Our work is part of the Engineering Information System (EIS) project, which is developing an object-oriented framework for integrating VLSI design and engineering tools and databases. The framework comprises an object model, FUGUE [HeZd88], (and a schema based on it) and an Object Management System. The framework provides a uniform interface to heterogeneous components (clients and servers) and serves as intermediary between clients that request data and services and servers that provide them.

The framework provides an object-oriented query language based on [Zdon88]. The underlying components of the federation provide storage and the implementations of query functions of the language. Note that implementations can span multiple servers; for example members of a particular set of objects might be stored and manipulated by different DBMSs. Also, servers are not limited to conventional DBMSs; some are embedded in engineering or administrative tools or suplied by file systems.

In FUGUE, sets are denotable objects. We model a multi-valued function as a single-valued function whose value is a set object. For example, if Employees is a function from departments to employees who work there, we say that the relationship between the toy department and the workers in that department Jack, Mary, and Pete is a single-valued function between the object Toy_Dept and the set object {Jack, Mary, Pete}. The result is that all relationships are one-to-one; an m-to-n relationship is modeled as a relationship between two set objects.

The identity of a set object is independent of its members. Sets are referenced by unique, immutable object identifiers (OIDs). All members of a set must be from the same domain. Sets with different element domains constitute different set types. For example, a set of objects of type X is of type Set_of_X. Sets have both set- and member-type-specific functions. Duplicate removal is provided by set-type-specific functions that test for member equality.

The separation of the identity of the set object from its composition allows us to express constraints about the sets themselves, e.g., that X is a subset of Y, rather than only about the particular content of X and Y. Moreover, objects X and Y ratain their identity even though their membership might change.

We provide multiple equality functions for set objects. Set identity (i.e., object identity) is based on OID identity. Other kinds of equality are based on the contents of the sets , i.e., "shallow-" and "deep-equality" as described in [KhCo86].

3. Inclusion graphs

Below, we describe an example of structures based on metadata about set objects that support one form of semantic query optimization. It is applicable in federated systems where the implementations of sets, i.e., both storage and manipulation functions, is provided by heterogeneous underlying servers whose performance characteristics are unknown by the global query processor.

We propose for certain sets to compute ancillary structures called "inclusion graphs" to capture set inclusion relationships. Use of the graph allows us to handle set comparison queries directly (to avoid accessing the set members individually). In addition, we can sometimes reduce the number of sets whose members must be examined individually, either because a particular set must necessarily satisfy the query or because it cannot possibly satisfy the query, based on inferences from the inclusion relationship.

For example, suppose we have a function that maps employees to courses they have taken. We can process the query "Find all employees who have taken at least all of the courses taken by employee x" by finding employees whose sets of courses-taken are supersets of the set of courses-taken by x. We can do this using the inclusion graph, without ever testing individual members of the sets of courses.

As another example, suppose we have a function that maps administrative units to the employees who work in them. To process the query "Find all administrative units in which at least one employee has taken CPR training," we can infer that when we have found such a unit, all units with a superset of its employees satisfy the query. Again, we can do it using the inclusion graph without testing any of the larger units.

We can define an inclusion graph for any set of objects S of type T that has a multi-valued function f defined on the members of S. We will view the values of the function f as objects that are instances of the type Set_of_T. The inclusion graph for a set S has a node corresponding to each distinct element in range(f). It has an edge (n1, n2) between each pair of nodes such that n1 is a superset of n2.

We can represent the sets over which we compute the inclusion graph by a bit vector which contains one bit for each member of the domain of elements of the set (in some predefined order). Membership of an element in a particular set is indicated by a one in the corresponding bit in the bit vector representation of the set. This provides a convenient structure for computing set inclusion as well as intersections and unions. Other representations may be more efficient for particular domains, e.g., where sets containing all (or nearly all) possible combinations of domain elements appear in the database.

Note that the inclusion relationship holds for sets in the domain S regardless of the function f. We can generalize the concept to build a single inclusion graph for a domain S even though multiple functions yield sets in S. This widens the class of queries that can be optimized based on the graph to include queries over values of each of the specified functions that yield sets in S. We can also use the graph to support queries that compare values of different functions that yield sets in S.

Maintenance of inclusion graphs must rely on triggers associated with the functions that create sets that are part of the inclusion graph or modify their contents. If multiple functions manipulate these sets, they require consistent triggers for maintaining the inclusion graph. It is likely impossible to retrofit such triggers once some functions that maintain the graph have been specified. For this reason, we might want to restrict the graph to references from a single function.

4. Future Work

It appears that the techniques described in [Agra89] provide an efficient mechanism for computing and maintaining inclusion graphs. We expect to apply these techniques and test them in our framework in the future.

We are also interested in the development of other structures that can support optimizations that are independent (or nearly independent) of the implementations provided by the components of a federated system. We expect to extend concepts of global optimization for systems that integrate heterogeneous databases and semantic query optimizations. We will base these extensions on other structural and behavioral information capturable by the object model.

In addition, we expect to develop a planner for our OMS that does base decisions on the characteristics of underlying processors and properties of the operations they perform.

Acknowledgements

The work on inclusion graphs that was described here has been a joint effort with Stan Zdonik.

References

[Agra89]
R. Agrawal, A. Borgida, H. Jagadish, "Efficient Management of Transitive Relationships in Large Data and Knowledge Bases," report submitted for publication, 1989.

[Chak88]
U. Chakravarthy, J. Grant, and J. Minker, "Foundations of Semantic Query Optimization for Deductive Databases,", Morgan Kaufmann Publications, 1988.

[HaZd80]
M. Hammer and S. Zdonik, "Knowledge-Based Query Optimization," Proceedings of the Conference on Very Large Databases, Montreal, Canada, 1980.

[HeZd88]
S. Heiler and S. Zdonik, "FUGUE, a Model for Engineering Information Systems and other Baroque Applications," Proceedings of the Third International Conference on Data and Knowledge Bases, Jerusalem, 1988.

[LaRo82]
T. Landers and R. Rosenberg, "An Overview of MULTIBASE," Distributed Databases (H.J. Schnieder, ed.), North Holland Publishing Co., Amsterdam, 1982.

[LiAb86]
W. Litwin and A. Abdellatif, "Multidatabase Interoperability," IEEE Computer, December, 1986.

[Nava86]
S. Navathe, R. Elmasri, J. Larson, "Integrating User Views in Database Design," IEEE Computer, January, 1986.

[Zdon88]
S. B. Zdonik, "Query Optimization in Object-Oriented Databases," Proceedings of the International Workshop on Object-Oriented Database Systems, Bad Munster, West Germany, Springer-Verlag, 1988.

Early experience with rule-based query rewrite optimization

Hamid Pirahesh
IBM Almaden Research Center
San Jose, CA 95120

Introduction

The functionality of relational database systems (DBMSs) must evolve to respond to the requirements imposed by new applications and changes in computing technology. In this paper, we describe the Starburst query rewrite optimizer, which transforms queries (and other SQL statements) roughly at the query language level. The Starburst plan optimizer may find plans for these revised queries that are more efficient than the plans it finds for the original query. Based on our early experience, rewrite optimization, which is becoming a significant part of query optimization, has many difficult open problems. We will describe the rule-based approach that we take in Starburst (using production rules, not grammar rules of [Lohm88]), which appears to be a promising approach to rewrite optimization, and we will describe some of the problems which we view as open.

The problem that optimization must handle is changing because the nature of the queries that DBMSs must handle is changing. The amount of data kept in large relational databases is expected to be in the terabyte range in the coming decade. The processing power of affordable parallel computers is expected to be over 1000 MIPS. That combination of data plus processing power creates the opportunity for much more complex queries. Hence we expect that future DBMSs will have to deal with applications which are increasingly data-intensive and logic-intensive. Common relational query languages such as SQL are not powerful enough to satisfy these requirements. For example, these languages have weaknesses in statistical analysis and structural (complex objects, record structures, etc.) expressibility, which are crucial for data summation and engineering databases, respectively. We expect that the functionality provided by query languages will grow considerably. More of the application logic will be moved inside the DBMS, both for better performance (bringing function to data) and for better sharing of data among applications (better protection of data by encapsulation). DBMSs will deal with a much larger set of data types and operations. From the application performance viewpoint, this is valuable since it allows more type specific operations to be specified in search predicates, so that (possibly massive) amount of irrelevant data does not pass through the DBMS to the applications. This is particularly significant since the data rate of the output of DBMSs is typically much less than the data rate of storage devices that they get data from. Operations such as outer join, recursion, and sampling should be handled by DBMSs for the same reason. A closely related area is support for object-oriented concepts, which should be included in the DBMSs for the same reasons (performance and encapsulation) cited above for data types. The functionality that queries express will grow significantly, so optimizing them will also become increasingly challenging.

Ad hoc interaction with DBMSs is commonly through high level user interfaces, allowing complex queries to be specified by users easily (where the users may not even be aware of the complexity of their requests). Often, a high level interface query results into many complex DBMS queries, which must have a short response time. This increases both the complexity and the traffic rate of DBMS queries. The same phenomenon occurs in interfaces between high level programming languages, such as Prolog, and DBMSs. These programming environments allow programmers to write applications resulting in many complex DBMS queries.

There are several other key areas, such as active databases, distributed/heterogeneous databases, and parallelism that require major support within DBMSs. Technological advances, such as large, inexpensive main memory and secondary storage, and high MIPS rate for CPUs affect the way DBMSs function. These advances affect what sort of requests optimization must handle and the trade-offs it must consider.

To recapitulate, we believe that the functionality of DBMSs, the amount of data they manage, and the traffic rate of queries they must handle are increasing rapidly. These put a major burden on query optimizers, which play a key role in the success of relational systems. Consequently we are facing a major problem in query optimization. Often, optimizers have blind points: they may not optimize some queries (or parts of queries) very well. This shortcoming is much more visible for very large databases because a small mistake in optimization may lead to orders of magnitude difference in response time and consumption of resources (e.g., I/O, CPU, memory, and communication), making query plans unacceptable despite the increase in MIPS and I/O bandwidth. Furthermore, increasing the number of operators (e.g., joins) and introducing complex new operators (e.g., recursion) increases the chance of dramatic optimization errors. Note that the space of alternatives grows exponentially with the number of operators (e.g. joins) in a query. We do not expect that exhaustive search will be acceptable, so heuristic-based optimization will be inevitable. Obviously, this increases the possibility that the optimizer will have blind points. Raising the level of functionality of query languages, as explained above, exacerbates the complexity of the optimization problem. Accurate selectivity estimates are crucial in optimization. It is not clear how we estimate the selectivities of predicates involving operators on abstract data types, or methods on objects. Hence optimizers must be **extensible** both for new functionality and so that they can be taught to avoid blind points.

DBMS parallelism makes the problem of query optimization even more complex. For parallelism, we usually partition the data (e.g., into 100 partitions) based on the predicates in the query, and run one task for each partition [DeWi86, Lori89]. In parallel hash join, a task does the join of the corresponding hash partition of the outer and the inner tables [Schn89]. Load balancing among tasks is very important. This requires choosing the partition sizes so that the tasks are balanced. This, in turn, requires estimating the amount of work associated with each partition. Usually, we do not have statistics for each partition, mainly because partitions are query dependent. We may have to rely on the overall statistics to estimate the work associated with each partition. It is expected that this estimate becomes increasingly inaccurate as the number of partitions grows, i.e., when increasing the degree of parallelism, or reducing the size of each partition. Hence, for a high degree of parallelism, we must find better ways of query cost estimation. We believe extensible optimization can play a major role in this area.

Experience with rule-based rewrite optimization

A major feature of rule-based systems is their flexibility, allowing extension of the system logic by specifying more rules. Hence, they seem to be a suitable base for building extensible optimizers. In Starburst we have taken this approach. The EXODUS project also has taken a rule-based approach [Graef87]. Here, we report on some of the advantages and disadvantages that we have experienced in our rule-based implementation (using production rules). Before doing this, let's quickly review the Starburst approach to query optimization.

We have divided the problem of query optimization into two parts: rewrite optimization and plan optimization. Plan optimization deals with low level operations such as choosing join order, join methods, and the access paths to tables. Rewrite optimization does query transformation for better performance. It does view merging, predicate pushdown, semantic query optimization ([King81]), etc. In a sense, Starburst uses a divide-and-conquer strategy, allowing each phase of query optimization to solve its part of the problem independently. Starburst is intended to support full SQL ([IBM87]) plus extensions to support the type of user requirements discussed above. As a result, optimization is a complex task, and decomposing it into two phases initially has helped us to simplify that task. We are investigating ways of integrating the two phases in a better way. In this report, we discuss our experience with the query rewrite optimizer. Details of the plan optimizer are discussed in [Lohm88, Lee88]. Rosenthal, et al. [Rose86] uses a transformation-based optimization similar to our query rewrite, for the entire query optimization problem.

In Starburst, we specify a set of rewrite production rules for rewrite optimization. The system has a rule engine, a library of rules, and an in-memory representation of the query being optimized (similar to the working memory in the OPS5 model [Brow85]). The in-memory representation of the query, called Query Graph Model (QGM), is a directed graph where the nodes (also sometimes called "boxes") correspond to operations on one or more tables, and the edges describe flow of data between boxes (i.e., the output of one box is used as an input table by another). Types of table operations include *Union*, *Intersection*, *Difference*, *Group By*, and *Select*. The *Select* operation can involve join, projection, quantification, and duplicate elimination. See [Haas88, Hasa88] for more details. System customizers may add new operations to the QGM. Examples of operations that might be added are sampling, statistical analysis, outer join, and heuristic searches. It is helpful to think of QGM as a flow graph, where tuples flow along the edges.

Rewrite rules describe transformations of QGMs that satisfy certain properties into semantically equivalent QGMs. Each rewrite rule has an IF part, which checks if the transformation can be done, and a THEN part, which does the transformation. For example, consider transformation of subqueries to joins in SQL [Kim82, Gans87, Hasa88]. The IF part of a rule checks to see if a subquery can be converted to join. The THEN part actually does the transformation. The current set of rules include subquery transformations (existential, universal, etc.), predicate pushdown towards sources of the data (in the sense of the flow graph), pushdown/pullup of duplicate elimination, table expression ([ANSI88, Haas88]) transformations, projection pushdown, and an extended version of magic set transformations. The IF part and the THEN part of the rules are each a procedure written in a procedural language (currently in the language C). The reason for this is that the rule language must be able to access and manipulate QGM in a complex way. Below we discuss several areas that need particular attention: **design** of the rules, **control** of execution of the rules, and **interference** between the rules.

Design of the rules

One problem we face is nonlinear growth of the number of rules when new operators are added to the query language. For example, for predicate pushdown, we need to specify under what conditions a predicate can move from one QGM box to another. These conditions depend on the types of the two boxes. As a result, whenever we add a new box type (e.g., for outer join), a naive implementation would have to specify rules for predicate pushdown from all the existing box types to the new box type, and vice versa. Hence, we need to specify at least 2 rules between any two box types. For a system with 20 box types, there might have to be 800 rules just for predicate pushdown! In fact, the problem is even worse, because there are different rules for predicates involving quantifiers (existential, universal, etc.) than for other predicates. This number of rules is not acceptable. Therefore, we designed predicate pushdown so that each rule is concerned with only one box type. For each (existing or new) box type, we specify the query transformation rules for predicates entering or leaving this box. To move a predicate from box A to box B, one rule specifies when and how the predicate can leave box A and another rule specifies how a predicate can enter box B. The rule for box A invokes the rules for box B before it commits to the pushdown of the predicate. Thus our rule system allows rules to be invoked not only by the rule engine, but also by other rules.

Control

Another problem we had to deal with was when to invoke a rule. We do not want rules to be fired whenever they are eligible; instead, we want to have more control on them, but not necessarily unlimited control. In OPS5, conflict resolution strategies (e.g., specificity or recency) are allowed; we need a more structured approach, for semantic clarity, performance, and extensibility. When we consider transforming a QGM box, we want to have as much information as possible about that box. Predicates (and duplicates) play a key role in determining what kinds of transformations are legal. For example, the set of predicates and the duplicate elimination attributes of a *Select* box may determine whether merge is possible, when *outer join* can be converted to regular join, and when joins are redundant. Therefore, we want to wait until all the rules associated with predicate pushdown and duplicates are applied to the boxes above a box (in the sense of the QGM of a query which is a directed acyclic graph) before applying the merge rules to that box. Control of the order of execution of the rules is even more critical for optimization of recursive queries using magic sets technique [Ullm85].

We do not want to have explicit general control of the rules, since that level of procedurality would destroy a major benefit of the rule-based approach. We introduce the concept of *contexts* explicitly. Roughly speaking, a context is a pointer to a QGM box. We establish a context and then we let the rule system apply all the rules applicable to that context. Therefore, each rule, as one of its input parameters, gets the current context. Control of the context is separated into another (extensible) component. In the context control component we traverse QGM and establish the context for the rules. In the transformation component, rewrite rules may be applied in the current context. A more abstract view of this problem is as follows. In optimization, control of the search space is critical. However, rule systems, such as OPS5, try to provide very limited control. This shortcoming is usually bypassed in an ad hoc way by introducing control variables in the working memory. We needed to have direct control of the search space.

Interference

A difficult problem in rule specification is interference between rules. Individual rules may seem clear, but they may interact in unexpected ways, especially in an extensible system. One major advantage of the rule-based approach is that it is easy to add new rules. Unfortunately, adding a new rule may require that existing rules be examined to determine how they interact with the new rule. Obviously this can be very difficult (or unacceptable administratively), but there may be subtle interactions that can be surprising (e.g., loops in the optimization of certain queries).

For example, assume the system already had the rules to eliminate redundant joins [Ott82], and rules using stored views were added. The new rules try to use stored views to filter the amount of information that must be obtained from the base tables. These rules can be advantageous in case the stored views are much smaller than their corresponding base tables because of column projection and tuple selection. Suppose V is a stored view containing all the monetary information (e.g., salary) about the employees and their departments for a particular division of a company. V is much smaller than the corresponding base tables because it has fewer columns and it has tuples just for a division, not for the whole company. Suppose a query wants *all* the employee information for employees working in that division subject to predicates on salary and location. The rule transforms the query so that it accesses the stored view V first to filter the employee tuples, and then accesses the employee table to get the rest of the columns and apply the rest of the predicates. Semantically, accessing the stored view V is redundant. After this transformation, the join elimination rules transform the query so that it does not access the stored view V because it is redundant, undoing what the stored view rule did. This may even loop because the stored view rule may again do the transformation, and repeat the cycle. Thus, when we add a rule, we must examine how it interacts with existing rules, and somehow prevent this undesirable interference.

Conclusion

Optimization continues to play a key role in future DBMSs. Current optimization technology is not adequate to handle the considerable functionality being added to DBMSs in response to user needs. Rule-based optimization is a promising approach, but still is immature. There is a need for much more intensive research in this area.

In Starburst, uniform representation of operations, predicates, etc. in QGM proved to be very valuable in reducing the task of introducing new operations and new rules. The generic structure of QGM allows the new rules to benefit from the many generic library routines introduced for existing rules.

Rule-based optimization proved to be valuable to provide extensibility. However, special attention must be made to design the rules to minimize redundancy between rules and maximize rule usability. Still more work has to be done to allow a more structured control of rule execution, and to have a better handling of rule interference.

Acknowledgements: The author would like to thank I. Mumick, G. Lohman, and particularly S. Finkelstein for careful reading and valuable comments.

Bibliography

[ANSI88] ISO_ANSI, Working Draft; Database Language SQL2, (August 1988).

[Brow85] Brownston, L., R. Farrell, E. Kant and N. Martin, Programming Expert Systems in OPS5, *Addison-Wesley* (1985).

[DeWi86] D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, M. Muralikrishna, Gamma-a high performance dataflow database machine, *Procs. of the 12th International Conference on Very Large Databases* (San Francisco, CA, Oct. 1986).

[Gans87] Ganski, R. and E. Wong, Optimization of Nested SQL Queries Revisited, *Procs. of ACM SIGMOD Conference* (1987).

[Grae87] G. Graefe, D.J. DeWitt, The EXODUS Optimizer Generator, *Procs. of ACM-SIGMOD* (San Francisco, CA, May 1987).

[Haas88] L.M. Haas, J.C. Freytag, G.M. Lohman, H. Pirahesh, Extensible Query Processing in Starburst, (1989). *Procs. of ACM-SIGMOD* (Portland, OR, May 1989).

[Hasa88] W. Hasan, H. Pirahesh, Query Rewrite Optimization in Starburst, *IBM Research Report RJ 6367* (1988).

[IBM87] IBM Systems Application Architecture, Common Programming Interface: Database Reference, *SC 26-4348-0* (Sept. 1987).

[Kim82] Kim, W., On Optimizing an SQL-like Nested Query, *ACM Trans. on Database Systems* **7:3** (Sept. 1982).

[King81] King, J., QUIST: A system for semantic query optimization in relational database, *Procs. of the Seventh International Conference on Very Large Databases* (1981).

[Lee88] M. Lee, J.C. Freytag, G.M. Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *Procs. of 14th Intl. Conf. on Very Large Data Bases (VLDB)* (Long Beach, CA, August 1988).

[Lohm88] G.M. Lohman, Grammar-like Functional Rules for Representing Query Optimization Alternatives, *Procs. of ACM-SIGMOD* (Chicago, IL, June 1988).

[Lori89] R. Lorie, J. Daudenarde, G. Hallmark, J. Stamos, H. Young, Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience, *IEEE Database Engineering quarterly Bulletin* (March 1989).

[Ott82] Ott, N. and K. Horlander, Removing Redundant Join Operations in Queries Involving Views, *Heidelberg Scientific Center, TR 82.03.003* (March 1982).

[Rose86] A. Rosenthal, Understanding and Extending Transformation-Based Optimizers, *IEEE Database Engineering quarterly Bulletin* (Dec. 1986).

[Schn89] D.A. Schneider, D. DeWitt, A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment, *Procs. of ACM-SIGMOD* (Portland, OR, May 1989).

[Ullm85] Ullman, J., Implementation of Logic Query Languages for Databases, *ACM Trans. on Database Systems* **10:3** (Sept. 1985).

Distributed Object-Oriented Query Optimization:

Perspectives, Problems, and Paradigms

Extended Abstract

Umeshwar Dayal[1]

Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square, Building 700
Cambridge, MA 02139
dayal@crl.dec.com

2 May 1989

[1]This work was done primarily while the author was at Xerox Advanced Information Technology [formerly, Computer Corporation of America, Research and Systems Division, Four Cambridge Centre, Cambridge, MA 02142].

1 Perspectives

The goal of any query optimizer is to produce an inexpensive access plan for evaluating a query expressed in some query language. The complexity of the optimization problem depends on the data model, the query language, the target language (in which the access plan is expressed), the strategy space considerd by the optimizer, and the execution environment (distributed, parallel, etc.). Object-oriented query optimizers are the latest stage in a series of progressively more complex optimizers that have tried to push the state-of-the-art in one or more of these dimensions.

Early work on query optimization focused on the relational data model and the select-project-join fragment of the relational algebra (roughly equivalent to a single aggregation-free SELECT block in SQL). The most important problems tackled by these optimizers were to find the optimal sequence of joins and to select the particular access paths and implementation algorithms for each relation and each operation in the query. In distributed database systems, additional complexities arose: the relations might be stored at different sites and might be fragmented. The important problems (in addition to selecting the optimal join sequence) were site selection (to join R and S, where R and S are at different sites, should we move R to S's site, S to R's site, or both to a third site?), distributed implementations of joins (transferring blocks of tuples across the network rather than single tuples at a time), distribution of a join across fragments, and load balancing. Also, additional tactics such as semijoins, which could potentially reduce the volume of data transferred across the network, were introduced. In distributed database systems that allow data to be replicated at several sites, there was the additional problem of choosing a physical copy for each operand relation (see papers in [14]).

Recently, the class of relational queries considered by the optimizer was extended to include nested queries, quantifiers, outerjoins, and aggregates [7]. The idea was to increase the power of the target algebra into which the query is compiled, and thus to enable new transformations to be applied to the query before access path selection. The extensions are especially relevant in a distributed environment because the transformed query potentially involves the transfer of less data (e.g., permitting nesting predicates to be treated as joins and evaluated by methods other than tuple substitution across the network).

Enhancing relational query optimizers to handle recursive queries has also received considerable attention in the literature, although the bulk of this work is for single-site queries only [1, 20].

Extending relational query optimization techniques to semantically richer data models (such as IFO or DAPLEX) required the solution of additional problems [3, 19]. First, these models permit inter-entity links (i.e., attributes whose values are references to entities) and queries can include these links. To build on the work on relational query optimization, these queries were mapped into an extended relational algebra, in which the inter-entity reference was treated as a special kind of join (a linked join). For example, the DAPLEX query: [for s in Student, for d in Dept(s), print (Name(s), Location(d)), endfor, endfor] was turned into a query that link-joined @Student and @Department on the condition [@Student.Dept = @Department.ID][2] (followed by projection). These linked joins may have special efficient implementations. At a single site, the links may be implemented by physical pointers. Across sites, however, they have to be implemented by symbolic pointers or by indirection through an entity directory

[2]We denote the relation corresponding to entity type E by @E

(which maps entity identifiers to physical addresses). A second problem was that these semantic models usually support set-valued attributes (e.g., Children: Employee -> set of Child). There were two alternatives for dealing with this problem: (i) treat the set-valued attributes as separate (normalized) relations (so, corresponding to Children, we would introduce the relation @Children with attributes EmployeeID and ChildID; then, the query [for e in Employee, for c in Children(e), print (Name(e), Age(c)), endfor, endfor] would require two linked joins among @Employee, @Children, and @Child); and (ii) complicate the target algebra with operators for non-first normal form relations (so the above query would be turned into a linked join between @Employee and @Child on the condition [@Employee.Children contains @Child.ID]). The choice we made for DAPLEX was the latter, because it more accurately reflected the special access paths provided in DAPLEX for implementing set-valued attributes. Third, because of the idiosyncracies of DAPLEX syntax, unidirectional joins turned out to be important, and we developed valid transformations for them (not all the transformations possible with regular joins work for outerjoins). Finally, for DAPLEX, we also had the problem that the language is procedural, so "queries" are actually written as programs involving iteration over entity sets (as in the above examples), conditionals, etc. To enable the use of extended relational query optimization techniques, these programs had to be "decompiled" into expressions (called "envelopes") in a target language to which algebraic transformations could be applied.

The next level of complexity for query optimization occurred in heterogeneous distributed database systems such as Multibase [15, 4]. Three problems were especially insidious. First, the various sites connected to Multibase might have vastly different capabilites (e.g., they might not be able to execute joins or semijoins, or to access temporary files). This required a description-driven approach, in which the capabilities of the individual sites were specified in a high-level language to the optimizer. Also, pieces of the access plan that were sent to a site had to be filtered to ensure that they could, indeed, be executed at that site, and then the missing pieces were subsequently compensated for (e.g., if a site was incapable of doing even a local join, then the operands had to be retrieved to some other site and the join had to be performed there). A second problem had to do with site autonomy: no site could have complete information about the access paths and implementation algorithms that other sites were going to use to process their pieces of the access plan. This led to a two-phased optimization approach: high-level, description-driven global optimization (to decompose the query into single-site subqueries), followed by detailed local optimization (during which each site selected access paths and implementation strategies, which could override some global decisions). Third, because the databases at different sites might contain inconsistent data, Multibase supported the definition of complex views. When queries were resolved against these views, complex queries involving (bidirectional) outerjoins and aggregates) typically resulted. The common tactic of moving selections and projections past joins and unions so that they can be locally processed does not generally work when aggregates and outerjoins are involved. New tactics were developed for this purpose.

Optimizing queries in object-oriented database systems subsumes these problems of optimization for relational and semantic data models, and introduces some new problems.

2 Problems

Optimizers for the relational model, and even for the semantic models, were "closed" in that they used a fixed set of tactics for optimizing a fixed repertoire of operators over a fixed set of access paths and implementation methods. The first new problem for object-oriented sytems is extensibility: users are

allowed to define arbitrary operators (methods) for each object type. Typically, the definer of the type can specify new access paths for objects of this type and can implement the operators using special hardware and software. The optimizer, therefore, has to be given enough knowledge about the types and their implementations. (Note that this is similar to, but significantly generalizes, the description-driven optimization problem considered for Multibase, where each site had a known subset of some fixed set of capabilities). The important issues relative to extensible query optimization are: what knowledge should be imparted to the optimizer? how should this knowledge be specified? how should the optimizer search the knowledge base for pertinent knowledge in constructing good strategies for a given query? A few recent papers describe first-cut solutions to these problems [2, 10, 12]. Note that even "closed" optimizers do not need to have complete information about how access methods (e.g., B-trees or hashed files) are implemented. The kind of knowledge useful to an optimizer includes the following: algebraic properties of the operators (these determine the valid transformations that can be performed on the query); cost information (e.g., execution times, result size); and properties of the input and output arguments of the operators that might affect processing (e.g., if the input arguments are sorted, the result might be, too; clustering or grouping is induced by some methods for computing projections and various flavours of join-like operations; the result of a projection- or grouping-like operation may or may not contain duplicates). In the first extensible optimizers, these properties were expressed via transformation rules. Search of the rule base was typically exhaustive and based on pattern matching. Extending these techniques to work in a distributed database system poses additional challenges: because the rule base may be distributed, some coordination among various sites is required to produce the global plan.

A second problem is how to deal with new, generic operators that appear to be common to many application domains, and hence have been proposed for inclusion in object-oriented data models. Examples of these from the PROBE project are traversal recursion and spatial join [8, 17]. Understanding the properties of these operators (e.g., does spatial join commute with regular join? are its results clustered? can selections and projections be moved ahead of or "inside" a traversal recursion?) and determining good strategies for queries that involve these operators, especially in a distributed system, remain open problems.

A general issue here is the class of queries expressible in the query language. Early object-oriented systems lacked powerful query primitives; instead, users wrote application programs to access one object instance, and then to navigate through the database following inter-object links. Recent object-oriented languages provide both navigational access (based on links) and associative access (based on attribute values); they also provide collection types and iteration operators so that both instance-at-a-time manipulation as well as collection-at-a-time manipulation is possible. The commonest collection type, of course, is the set (as in DAPLEX), but other types (such as sequences, lists, bags, and arrays) have been posited. For DAPLEX, we had studied how to optimize queries with mixed navigational and associative elements. Also, decompilation was a way of optimizing mixed instance and set queries. Clearly, more work is necessary to extend these techiques to work with collection types other than sets. Furthermore, since the trend is to seamlessly integrate querying primitives into programming languages (e.g., rather then invent new iterators for entity sets, DAPLEX was deliberately designed to blend into ADA syntax [Smith83]), it would be interesting to explore the integration of compiler optimization techniques with query optimization techniques. A start in this direction appears to have been made for the E persistent programming language on the EXODUS project [18].

Several researchers have pointed out that a major problem in object-oriented query optimization is the

conflict between encapsulation and optimization [5, 11]. A fundamental tenet of the object-oriented approach is that users of an object "see" only the interface (i.e., the specification of its type and the signatures of its operators); the details of how the objects are represented and how the operations are implemented are hidden. However, if this information is hidden from the optimizer as well, then clearly there is no hope for optimization. In general, it may not even be possible to predict which other methods and objects may be invoked from within a method that is explicitly referenced in the query, or even at which sites these other objects and methods are located. Several approaches to overcoming this problem have been suggested. The simplest is to define a second interface to objects, through which the optimizer can access the kind of knowledge it needs for extensible query optimization [5]. In fact, any number of such interfaces (or views) can be defined for each object type [13]. A more sophisticated proposal is to introduce a "revelation" phase in which each object type (or each object) involved in the query has the option of revealing this information in the form of an algebraic expression that can be subjected to optimization [11].

The site selection problem is exacerbated in object-oriented systems, because there is the additional complexity of deciding whether to move the object to the method, the method to the object, or both to some third site. Clearly, a prerequisite is that the method can in fact be executed at the selected site. In homogeneous, distributed systems, each site can execute the same set of methods, and already has the "code" for all these methods. In a heterogeneous system such as Multibase, each site was assumed to be a server, capable of executing some fixed (but possibly different from other sites) set of methods. This same simplifying architectural assumption could be made for distributed, object-oriented systems as well.

Finally, there is the "complex object" problem. It has been argued elsewhere that queries over a complex object might spawn a collection of queries over the components of the complex object [6, 11]. Typically, the queries in the collection are related. (For example, the query to retrieve part assembly X might require the retrieval of all components connected via various relationships to assembly X.) Simultaneously optimizing this collection of queries is an interesting problem. How the complex object and its components are stored (e.g., whether they are clustered or not), and the availability of hierarchical access methods such as indices (to aid the retrieval of the complex object and its components), could dramatically affect the cost of processing. To enable the optimizer to consider efficient access plans over hierarchical access methods, the target language should include hierarchical operators. Note that outerjoins, which were important for DAPLEX and Multibase, may be interpreted as producing hierarchical results, but we additionally need hierarchical selection and projection operators (like the ones in [9]). However, because our model also allows non-hierarchical relationships, and allows objects to participate in multiple relationships, clustering or other hierarchical access methods are not always feasible. In the absence of such access methods, it might be more effective to use multiple query optimization techniques to simultaneously optimize the collection of queries spawned by a complex object query [21, 22]. Hence, the optimizer must be capable of producing both types of access plans (involving hierarchical operators and multiquery strategies).

3 Paradigms

We postulate a multiple server architecture for distributed, object-oriented database systems. Each server stores one or more classes of objects, and implements operations (methods) of these classes. We assume that each server is localized to a single site, although there may be more than one server per site, and the same type of service may, in fact, be provided by servers at more than one site. Query

optimization is performed by agents that may be local to a single server (i.e., have knowledge of how to optimize queries involving operators over the object classes supported by the server), local to a single site (e.g., Multibase's local query optimizers), or global (e.g., Multibase's global optimizer or Sytem R*'s distributed query compiler [16]). Thus, these optimizer agents may be specialized in various ways: for spatial queries, for recursion, for integration operations (e.g., outerjoins), for multiple query optimization, and so on. This architectural paradigm addresses the extensibility, encapsulation, and autonomy requirements for distributed, object-oriented database systems.

Optimizing a query will, in general, require coordination among several optimizer agents. A global agent at the site where a client's query is posed may be the natural choice for coordinator. Various paradigms for coordination may be useful. Instead of assuming that all information needed for global optimization is available at the coordinator (as in Multibase), we want to support negotiation. The coordinator constructs an initial global plan, based on knowledge that it has about other agents and servers. It broadcasts (or selectively multicasts) a piece (one operation or a subplan) of the plan; other agents respond with "bids" summarizing (in a high-level language) their best strategies for processing that piece of the plan, and their best costs. These bids will, presumably, be derived by local optimization, based on knowledge that the agents have about the access methods, implementation algorithms, etc., of the local servers, the load at the local site, and other relevant factors. These bids are used by the coordinator to refine its global plan (e.g., to apply algebraic transformations to the plan, to do site selection for various operations in the plan, and to select a particular physical copy of a replicated object and a particular server to provide a service); and so on.

Another relevant paradigm is that of collaborative planning. Instead of assuming that there is a single, undifferentiated collection of rules (as in the first generation of extensible optimizers), we think of each agent as an expert in solving some part of the optimization problem. (In particular, the global optimizers are experts in coordination and in conducting negotiations.) Then, under the control of a coordination expert, the various participating experts (selected, perhaps, via negotiation) cooperate to solve the problem of optimizing the whole query. For example, the query "Find all plastic parts and subassemblies in assembly X that are located within 5 centimetres of a pipe carrying a hot fluid" may require the cooperation of a spatial expert (to compute the distance of a part from a pipe), a selection expert (to select plastic parts and pipes carrying hot fluid), a domain expert (to interpret the predicate "hot fluid"), a hierarchical query expert (to retrieve the complex objects, viz., assemblies and component parts; this expert may have come in with a lower bid than the multiple query expert), a traversal recursion expert (to recursively compute the locations of parts within assemblies), and an integration expert (if data about part assemblies is distributed over several servers).

These novel paradigms for query optimization generalize various existing and proposed optimization techniques such as Multibase's two-phased optimization, System R*'s distributed compilation, rule-based extensible query optimization, and revelation. Several research problems have to be solved before this approach can be realized in a working system. These include: language(s) for expressing rules used by the various agents; the target language for expressing plans; protocols for negotiation and cooperation; interfaces between the agents in support of these protocols; and the details of rules used by the coordinator and by specialized agents. We believe that this is a promising, but challenging, prospectus for research in query optimization in distributed, object-oriented database systems.

[14] W. Kim, D. Reiner, D. Batory (editors).
Query Processing in Database Systems.
Springer-Verlag, 1985.

[15] T. A. Landers, R. L. Rosenberg.
An Overview of Multibase.
Distributed Databases.
North Holland, 1982.

[16] G. Lohman, et al.
Query Processing in R*.
Query Processing in Database Systems.
Springer Verlag, 1985.

[17] F. Manola, J. Orenstein.
Toward a General Spatial Data Model for an Object-Oriented DBMS.
In *Proceedings, Twelfth VLDB.* 1986.

[18] J. E. Richardson, M. J. Carey.
Persistence in the E Language: Issues and Implementation.
Technical Report 791, Computer Science Department, University of Wisconsin-Madison, September 1988.

[19] D. Ries, A. Chan, U. Dayal, S. Fox, K. Lin.
Decompilation, Optimization, and Pipelining for Adaplex: A Procedural Database Language.
Technical Report CCA-82-04, Computer Corporation of America, 1983.

[20] A. Rosenthal, S. Heiler, U. Dayal, F. Manola.
Traversal Recursion: A Practical Approach to Supporting Recursive Applications.
In *Proceedings, ACM SIGMOD Conference.* 1986.

[21] A. Rosenthal, U. S. Chakravarthy.
Anatomy of a Modular Multiple Query Optimizer.
In *Proceedings, Fourteenth VLDB.* 1988.

[22] T. K. Sellis.
Global Query Optimization.
In *Proceedings, ACM SIGMOD Conference.* 1986.

References

[1] F. Bancilhon, R. Ramakrishnan.
An Amateur's Introduction to Recursive Query Processing Strategies.
In *Proceedings, ACM SIGMOD Conference.* 1986.

[2] D. Batory, T. Y. Leung, T. E. Wise.
Implementation Concepts for an Extensible Data Model and Data Language.
ACM Transactions on Database Systems 13(3), September, 1988.

[3] A. Chan, U. Dayal, S. Fox.
An ADA-Compatible Distributed Database Management System.
Proceedings, IEEE 75(5), May, 1987.

[4] U. Dayal.
Query Processing in a Multidatabase System.
Query Processing in Database Systems.
Springer-Verlag, 1985.

[5] U. Dayal, et al.
PROBE -- A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis.
Technical Report CCA-85-03, Computer Corporaton of America, 1985.

[6] U. Dayal, et al.
Simplifying Complex Objects: The PROBE Approach to Modelling and Querying Them.
Datenbanksysteme in Buero, Technik, und Wissenschaft.
Springer Verlag, 1987.

[7] U. Dayal.
Of Nests and Trees: A Unified Approach to Processing Queries that contain Nested Subqueries, Aggregates, and Quantifiers.
In *Proceedings, Thirteenth VLDB.* 1987.

[8] U. Dayal, M. DeWitt, D. Goldhirsch, J. Orenstein.
PROBE Final Report.
Technical Report CCA-87-02, Computer Corporation of America, 1987.

[9] A. Deshpande, D. Van Gucht.
An Implementation of Nested Relational Databases.
In *Proceedings, Fourteenth VLDB.* 1988.

[10] G. Graefe, D. DeWitt.
The Exodus Optimizer Generator.
In *Proceedings, ACM SIGMOD Conference.* 1987.

[11] G. Graefe, D. Maier.
Query Optimization in Object-Oriented Database Systems: A Prospectus.
Advances in Object-Oriented Database Systems.
Springer Verlag, 1988.

[12] L. Haas, J. Freytag, G.Lohman, H. Pirahesh.
Extensible Query Processing in Starburst.
In *Proceedings, ACM SIGMOD Conference.* 1989.

[13] S. Heiler, S. Zdonik.
Views, Data Abstracton, and Inheritance in the FUGUE Data Model.
Advances in Object-Oriented Database Systems.
Springer Verlag, 1988.

Expression Processing in Iris

Waqar Hasan, Peter Lyngbaek, Kevin Wilkinson
lastname@hplabs.hp.com
Hewlett-Packard Laboratories

1 Introduction

The Iris object-oriented database system being developed at Hewlett-Packard Laboratories [FBC+87] is intended to meet the needs of new and emerging database applications such as office information and knowledge-based systems, engineering test and measurement, and hardware and software design. These applications require a rich set of capabilities that are not supported by the current generation of relational DBMSs. In addition to the usual requirement for persistence of data, associative access, controlled sharing, reliable storage, and efficient access, the new capabilities that are needed include: rich data modeling constructs, including constructs for modeling of behavior, direct database support for inference, novel and extensible data types, and multiple versions of data.

In order to meet the above needs, Iris is based on a semantic data model [LK86, FBC+87] that supports abstract data types. Object and function concepts provide high-level structural as well as behavioral abstractions. Functions in Iris may be implemented in three ways: *stored*, *derived*, and *foreign*. A relational storage manager maintains the extensions of stored functions in tables. Derived functions are defined by an Iris expression and foreign functions are defined by programs written in conventional programming languages. Functions with side-effects are called procedures. The roots of the Iris model can be found in previous work on Daplex [Shi81] and the Taxis language [MBW80]. A number of recent data models, such as PDM [MD86] and Fugue [HZ88], also share many similarities with the Iris data model. Iris queries are written as functional expressions and function values can be modified by database procedures. Extensibility is provided by allowing users to define new functions.

2 Expression Processing in Iris

An Iris expression (query or update) is a functional expression tree, or *F-tree*, consisting of function calls, variables, and constants. Such an expression is processed by translating it into a form which is then interpreted. The translation process consists of three main steps. First, the F-tree is converted to a canonical form. This involves a series of tree transformations that are done to simplify subsequent transformations. For example, nested function calls are unnested by introducing auxiliary

variables. Type checking is also performed. The actual arguments in a function call must match or be subtypes of the corresponding formal argument types.

The second step converts the canonical F-tree to an extended relational algebra tree known as an *R-tree*. This is a mechanical process in which function calls are replaced by their implementations (which are, themselves, R-trees). For example, comparison function calls (e.g. equal, not-equal, less-than, etc.) are replaced by relational algebra filter[1] operators. The logical function, *And*, is converted to a cross-product operator.

The resulting R-tree consists of nodes for the relational algebra operations of project, filter and cross-product. To increase the functionality of the relational algebra interpreter, there are additional nodes. A *temp-table* node creates and, optionally, sorts a temporary table. An *update* node modifies an existing table. A *sequence* node executes each of its subtrees in turn. A *foreign function* node invokes the executable code that is the implementation of a foreign function.

The leaves of an R-tree actually generate the data that is processed by the other nodes. A leaf may be either a table node or a foreign function node. A table node retrieves the contents of a storage manager table. A projection list and predicate can be associated with the table node to reduce the number of tuples retrieved. A foreign function node simply invokes a foreign function.

The semantics of the tree are that results of a child node are sent to the parent node for subsequent processing. For example, a project node above a table node would filter out columns returned by the project node. Joins are specified by placing a filter node above a cross-product node to compare the columns of the underlying cross-product.[2]

The final, and most complex, step is to optimize the R-tree. The optimizer is rule-based. Each rule consists of a test predicate and a transformation routine. The test predicate takes an R-tree node as an argument and if the predicate evaluates to true, the transformation routine is invoked. The predicate might test the relative position of a node (e.g., filter node above a project node) or the state of a node (e.g., cross-product node has only one input). The possible transformations include deleting the node, moving it above or below another node, or replacing the node with a new R-tree fragment. As in [DG87], the optimizer must be recompiled whenever the rules are modified.

Rules are organized into rule-sets which, together, accomplish a specific task. For example, one rule-set contains all rules concerned with simplifying constant expressions (e.g., constant propagation and folding). Optimization is accomplished by traversing the entire R-tree for each rule-set. During the traversal, at a given node, any rule in the current rule set may be fired if its test predicate is true.

[1] In order to distinguish the Iris system function, `Select`, from the select operator of the relational algebra, the term, *filter operator*, will be used to denote the latter.

[2] Of course, joins are rarely executed this way because the filter predicate is typically pushed down below the cross-product to produce a nested-loops join.

The optimization steps (i.e., rule-sets) can be roughly described as follows. There is an initial rule set that converts the R-tree to a canonical form. The canonical form consists of a number of expression blocks. An expression block consists of a project node above a filter node above a cross-product node (any one of these nodes is optional). A leaf of an expression block may be either a table node, a foreign function node or another expression block. An expression block may be rooted by a temp-table node, an update node or a sequence node.

A second rule-set eliminates redundant joins. This has the effect of reducing the number of tables in a cross-product. A third rule set is concerned with simplifying expressions. A fourth rule set reorders the underlying tables in a cross-product node to reduce the execution time by taking advantage of indexes. A fifth rule set handles Storage Manager-specific optimizations, for example, finding project and filter operations that can be performed by the Storage Manager.

The final (optimized) R-tree is then sent to the relational algebra interpreter which processes the R-tree and returns the result to the user. However, if the R-tree defines the body of a derived function it is simply stored in the database system catalog. The R-tree may be retrieved later when compiling expressions that reference the derived function.

The expression translator is flexible and can accommodate any optimization that can be expressed in terms of a predicate test on a node and a tree transformation. Of particular interest is the ability to optimize the usage of foreign functions. As a simple example, given a foreign function that computes simple arithmetic over two numbers, rules could be written to evaluate the result at compile time if the operands are constants. The Iris optimizer is further described in [DS89].

3 Issues in the Design of the Iris Expression Translator

Based on our experience with the Iris expression translator, which has been functional for more than two years, we expect to focus our research effort on the following areas.

Optimizing Functional Rather than Relational Expressions

We plan to explore the feasibility of evaluating functional expressions directly, thus eliminating the need for translating the expressions to extended relational algebra and then evaluating the relational expressions. This involves developing a new execution model for the functional expressions, possibly incorporating function caches. The expression processor must dynamically decide to process the functional expression directly, process the entire expression as a relational expression, or perhaps split the expression into several parts, some of which are evaluated as functional expressions and others as relational expressions. Such decisions are complicated by the fact that data in the function value cache may be out of sync with corresponding

data in the database.

Optimization of Foreign Functions

The Iris data model allows users to introduce functions that are implemented by programs written in conventional programming languages. When such functions are invoked as part of a query, the corresponding programs are dynamically loaded and executed. Currently, the query optimizer has no special knowledge about such functions except that input arguments must be instantiated before the function can be invoked; see the following research area. Therefore, the optimizer cannot generate optimal plans for queries that involve foreign functions. As part of defining a foreign function, it should be possible for the definer to add rules to the optimizer's rule base to indicate how usages of the function can be best optimized.

Extensibility of Optimizer

As described above, a major part of the Iris expression translator is rule-based. In order to introduce new optimization techniques or new constructs in the database language, the rule-base needs to be extended. This has proven to be a difficult task, due to close interrelationships between rules. For example, one rule set in Iris may generate a join order for nested loop joins by placing relations with indexes as the inner relations. Another rule set may generate a join order by placing relations corresponding to foreign functions such that the functions' input arguments are guaranteed to be instantiated before the functions are called. Defining the two rule sets separately may be simple, but combining them to attain both goals has proven difficult.

Side-Effects

The relational algebra used in Iris expression translation has been extended with operators for function updates and sequences of such updates. A sequence of function updates that access different data elements physically stored in the same tuple can potentially be combined into a single update operation. We plan to further extend the algebra with operators for object creation and deletion. We need to understand how such relational agebra expressions can be optimized. We expect that compilation techniques developed for programming languages can be applied.

Storage Manager Interface

Currently the Iris expression translator generates evaluation plans for a specific storage manager. Actually, the code of the translator makes assumptions about the storage manager, thus complicating the task of replacing the storage manager. In the long run we would also like to simultaneously use multiple storage managers, each specialized for a different purpose, e.g. text, spatial information, business data.

Two challenges we face are to define a "generic" storage manager interface and to optimize expressions whose evaluations require access to several storage managers.

References

[DG87] D. J. Dewitt and G. Graefe. The EXODUS Optimizer Generator. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 160–172, 1987.

[DS89] N. Derrett and M. C. Shan. Rule-Based Query Optimization in Iris. In *Proceedings of ACM Annual Computer Science Conference*, Louisville, Kentucky, February 1989.

[FBC+87] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

[HZ88] S. Heiler and S. Zdonik. Views, Data Abstraction, and Inheritance in the FUGUE Data Model. In Klaus Dittrich, editor, *Lecture Notes in Computer Science 334, Advances in Object-Oriented Database Systems.* Springer-Verlag, September 1988.

[LK86] P. Lyngbaek and W. Kent. A Data Modeling Methodology for the Design and Implementation of Information Systems. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[MBW80] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems*, 5(2), June 1980.

[MD86] F. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[Shi81] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), September 1981.

Query Optimization Basics for Object-Oriented Databases

Position Statement for Workshop on Database Query Optimization

S. L. Osborn

Dept. of Computer Science

The University of Western Ontario

London, On. Canada N6A–5B7

May 9, 1989

Object-oriented databases are characterized by complex data structures, specialized operations and object identity which persists. The following assumptions are made in this position statement:

1. The possible structures which database objects have are rich and varied. At the very least, there are sets of objects and aggregates or tuple-like structures, of which the elements or components can also be complex structures. An application may deal with collections or sets of objects, or with single, highly complex structures like a design. Only one data structure was offered by the relational model – sets of tuples. In general, an object-oriented database has more data structures.

2. Object identity is part of the data model and the user's perception of the data. In the relational model, all connections between tuples are made by having equal values appear in different relations. In an object-oriented system, two employee tuples can have the identical address object as one of their attributes. Thus connections may be made by finding identical sub-objects in two objects. We feel that as well as identity comparisons, equality comparisons should also be allowed and supported in any query interface.

3. Since this is a database, querying and data manipulation take place. Querying can be done by browsing through a complex object, or by taking subsets of sets of objects. Data manipulation, on the other hand, involves taking bits and pieces of objects and putting them

together in new ways to create new objects. We believe both kinds of activities should be supported, although in some existing proposals for object-oriented database systems, only the browsing activities are supported. Data manipulation was what set the relational model apart from the record-at-a-time databases. Powerful data manipulation should also be a part of object-oriented database systems. Actually, even an operation like the relational algebra projection creates objects which are not currently in the database.

Assuming that non-trivial data manipulation is allowed in an object-oriented database, and that the system can distinguish such operations as database operations, query optimization becomes an issue. To begin with, rearranging a sequence of such operations to yield a more efficient sequence should be considered. In order to do this, one needs a notion of when two sequences of operations give "the same" answer. Since object-oriented databases have a notion of object identity, two queries might yield answers which are *identical*, which means, they are same object, i.e. they have the same object ID. In other cases, two queries might produce results which are *equal*, which means have the same values but not necessarily the same object IDs. A third case must also be considered: two queries might produce a set (or a tuple) of objects, the members (components) of which are identical, but for which the object IDs of the set (tuple) itself are different. We call this type of "sameness" *shallow identity*. This use of equal is the same as the deep–equal of [KC86], and shallow identity here corresponds to their shallow–equal.

We have been developing an object algebra for use with an object-oriented database system [Osb88b, Osb88a]. With an algebra, it is possible to consider rearranging the operations to give an alternative execution order. Some of our findings regarding this type of query optimization in an object algebra are [Osb89]:

1. It is important to distinguish among equality, identity and shallow identity when discussing this type of query optimization.

2. For queries involving taking subsets of a set of objects (like the relational algebra Select), and set operations like Union, Intersect and Subtract, shallow identity can be guaranteed when the algebra operations are rearranged. One might think this is a negative result, because identity is not guaranteed. However, users probably do not care about the object ID of the set containing the objects they have retrieved, but would be content to know that the same objects are in the set.

3. For queries which restructure the data, or build any new objects, such as would occur with the relational algebra Project, the best we can expect is equal results, and occasionally when working with singleton aggregates or tuples, shallow identity. This is not very surprising, since new objects are being created. It is useful to know in what circumstances equality can be guaranteed.

4. Single objects should not be overlooked in query optimization for object-oriented databases. Relational databases deal only with sets of tuples, so all the optimization results deal with sets. Object-oriented databases are meant to deal with objects, which may be single, albeit deeply nested, objects. Many applications suggested for object-oriented databases deal with design, either in an engineering sense or of software. In these applications, users may well be dealing with a single object – the design. We have shown that sometimes with a single aggregate, shallow identity can be guaranteed in situations where only equality would hold with sets.

5. It makes no difference, for this kind of query optimization, whether we are working with strongly typed sets or untyped sets. It will make a difference when building indices or other specialized access paths.

References

[KC86] S.N. Khoshafian and G.P. Copeland. Object identity. In *OOPSLA '86 Proceedings*, pages 406–416. ACM SIGPLAN Notices, vol. 21, no. 11, Nov. 1986.

[Osb88a] S.L. Osborn. Identity, equality and query optimization. In K.R. Dittrich, editor, *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Datbase Sytems*, pages 346–351. Springer-Verlag Lecture Notes in Computer Science 334, Sept. 1988.

[Osb88b] S.L. Osborn. Polymorphism in an object-oriented database. submitted for publication, Sept. 1988.

[Osb89] S.L. Osborn. Algebraic query optimization for an object algebra. in preparation, 1989.

Efficient Querying in an Object-Oriented Database[†]

Extended Abstract

Gail M. Shaw
Stanley B. Zdonik

Department of Computer Science[‡]
Brown University
Providence, R.I. 02912

Abstract

We are interested in efficiently accessing data in an object-oriented database. We have developed a query algebra [Sha89a], which forms a base for our current research in query optimization, and have identified a variety of query transformations on that algebra. Our current work involves estimating the cost of such transformations.

1 Introduction

A major aspect in the development of query algebras is the potential for optimization. In the next section we outline an algebra that synthesizes relational query concepts with object-oriented databases. The algebra supports an object-oriented model with abstract data types, type inheritance, encapsulation, and object identity. Unlike other languages proposed for object-oriented databases (e.g. [Ban87], [Mai87], [Ban88], [Car88], [Osb88]) our algebra fully supports these object-oriented concepts and still provides full associative access to the database, including a unique join capability. The similarities between the structure of the algebra and the relational algebra lead us to believe that relational optimization results will prove useful in object-oriented query optimization. In addition, our consistent approach to abstract data types and support for object identity should offer flexibility in applying a variety of optimization strategies.

Query algebras can support optimization through syntactic transformations (e.g.[Osb88], [Ban87]) and we have found our algebra to be conducive to such transformations (see Section 3). However, the implementation of abstract data types involves the encapsulation of data and operations, requiring optimization strategies for encapsulated behaviors (e.g. [Gra87]). We present some of our current thoughts on optimization for object-oriented databases in Section 4.

[†]Support for this research is provided by IBM under contract No. 559716, by DEC under award No. DEC686, by ONR under contract N0014-88-K-0406, by Apple Computer, Inc., and by US West.

[‡]Electronic addresses: gms and sbz @cs.brown.edu.

2 The Encore Query Algebra

Our query algebra is based on the ENCORE object-oriented data model (see [Zdo86] and [Sha89b]). The model includes abstract data types, type inheritance, typed collections of typed objects, and objects with identity. We query over collections of objects using the type of objects in the collection as a scheme for the class. The collections are considered to be homogeneous, although the objects in the collection may have a type which is a subtype of the collection type.

The algebraic operations can be divided into two categories:

1) operations that retrieve data: Select, Image, Project, Ojoin, Union, Intersection, and Difference

2) operations that support data retrieval through manipulation of result structure and identities: Flatten, Nest, UnNest, DupEliminate, and Coalesce

The first four operations are defined in Table 1. The *Select* operation creates a collection of database objects which satisfy a selection predicate. The *Image* operation is used primarily to return a single component, or value, for each object in the queried class. In some ways, this is a selection of objects from a different collection than the one over which the query is posed.

The Project and Ojoin operations create new relationships not explicitly defined in the object types. These operations produce tuples to store the computed relationships. The *Project* operation creates one tuple for each object in the collection being queried, with the tuple storing selected relationships between components of the object or relationships involving database objects from other collections. *Ojoin* is an explicit join operator used to create relationships between objects from two collections in the database. In the Ojoin definition in Table 1 we assume that the two collections contain objects having user-defined data types (i.e., not tuple data types). The Ojoin operation then creates 2-tuples, storing one object from each collection in each attribute. When Ojoin involves a collection of n-tuples, the result is an (n+1)-tuple; i.e. tuples will not be nested within tuples by the operation. Although Ojoin is a special case of Project (see Table 2) the redundancy should be useful for optimization. In particular, our definition of Ojoin is designed to preserve the associativity of the operation.

The algebraic operators preserve the typing of the object-oriented data model and maintain the identities of objects in the database. The details of the operations are sensitive to different notions of object equality, which may prove useful in optimization. For example, DupEliminate provides the option of eliminating, from a collection, objects which have separate identities but equal (by some definition) values. Coalesce allows manipulation of

$$Select(S,p) = \{s \mid (s \text{ in } S) \wedge p(s)\} \quad (1)$$

$$Image(S,f) = \{f(s) \mid s \text{ in } S\} \quad (2)$$

$$Project(S, <(A_1,f_1),...,(A_n,f_n)>) = \{< A_1 : f_1(s),...,A_n : f_n(s) > \mid s \text{ in } S\} \quad (3)$$

$$Ojoin(S,R,p) = \{< A_s : s, A_r : r > \mid s \text{ in } S \wedge r \text{ in } R \wedge p(s,r)\} \quad (4)$$

Table 1: Some Query Operations

$$Ojoin(As, Bs, A, B, \lambda a \lambda b\ p(a,b)) \equiv Select(UnNest(Project(As, \lambda a < (A,a),(B,Bs) >), B), \lambda t\ p(t.A, t.B)) \quad (1)$$

$$Coalesce(S, A_k, i) \equiv Project(Ojoin(S, NoDupA_k, , K, \lambda a \lambda b\ a.A_k =_i b), \lambda t < (A_1, t.A_1), \ldots, (A_{k-1}, t_{k-1}), (A_k, t.K), (A_{k+1}, t.A_{k+1}), \ldots, (A_n, t.A_n) >) \quad (2)$$

$$\text{where} NoDupA_k := DupEliminate(Image(S, \lambda s\ s.A_k), i)$$

$$Select(Select(S, p_1), p_2) \equiv Select(Select(S, p_2), p_1) \quad (3)$$

$$Select(Select(S, p_1), p_2) \equiv Select(S, p_1 \wedge p_2) \quad (4)$$

$$Union(Select(S, p_1), Select(S, p_2)) \equiv Select(S, p_1 \vee p_2) \quad (5)$$

$$Select(Ojoin(A, B, p), \lambda t\ p_s(t.A)) \equiv Ojoin(Select(A, \lambda a\ p_s(a)), B, p) \quad (6)$$

$$Select(Ojoin(A, B, \lambda a \lambda b\ p(a,b)), \lambda t\ p_s(t.A, t.B)) \equiv Ojoin(A, B, \lambda a \lambda b\ p(a,b) \wedge p_s(a,b)) \quad (7)$$

$$Ojoin(A, B, \lambda a \lambda b\ p(a) \wedge p'(a,b)) \equiv Ojoin(Select(A, p(a)), B, \lambda a \lambda b\ p'(a,b)) \quad (8)$$

Table 2: Some Algebraic Identities

a result structure to provide aliasing of "equal" objects. Both of these operations could be used to create result structures which might be more efficient for querying.

3 Query Transformations

We expect the structure of the operations and the redundancies in the operator set to offer many opportunities for optimization. Some redundant algebraic operations are noted in Table 2 (identities 1 and 2). The equivalent operations return the same objects from the database, although the objects in which the results are stored have distinct identifiers.

We have also applied some relational optimization results to the object-oriented query algebra. For example, in Table 2 we note that selection predicates can be applied in any order (identity 3) and can be combined (identities 4 and 5), as in relational algebra. Identity 6 is equivalent to the relational optimization strategy of pushing Selection past Join. Similarly, when a Select operation is composed with an Ojoin, it may be possible to instead compose the two predicates to produce a single operation (identity 7). These two ideas are combined in identity 8; if an Ojoin predicate contains a conjunct involving only one of the operand classes, that conjunct can be extracted from the Ojoin predicate to form a Select predicate on the appropriate operand.

Another optimization consideration for object-oriented databases is the creation of new objects to store intermediate query results. The Project operation should prove useful in such optimizations by allowing the extraction, from objects, of information needed by subsequent operations. For example, consider the query $Ojoin(S, Q, \lambda s \lambda q\ s.length = q.length)$ which creates pairs of stacks and queues with equal lengths. The join could involve multiple accesses to each stack and queue to compute the length information. However, the Ojoin could be transformed into the following sequence of operations:

```
ST       := Project(Stacks,λr <(S,r),(LenS,r.length)>)
QU       := Project(Queues,λr <(Q,r),(LenQ,r.length)>)
Temp     := Ojoin(ST,QU,λs λq s.LenS = q.LenQ)
SQPairs  := Project(Temp,λt <(S,t.S),(Q,t.Q)>)
```

These operations accomplish the same result with only one evaluation of the length property

for each object in the collections. The information required for the join (i.e., the length for the predicate and the objects themselves for the result) is extracted once for each object and stored in tuples (in collections ST and QU).

4 Thoughts on Cost Estimation

In order to use the algebraic transformations effectively we need to identify a means for estimating the cost of a query. This involves developing a definition of query cost, as well as a method for evaluating the cost.

The cost measure will probably be related to the number of object accesses needed to respond to the query. This, in turn, can depend on the query itself, the selectivity of the collection(s) being queried, the implementation of the collection being queried, and the type and implementation of the objects in the collection. The structure of objects implies that access to a single object could additionally involve access (recursively) to objects it references. The cost, in objects, of accessing a single object would be largely determined by the query, although the disk access costs of retrieving different levels of the object could also depend on the storage implementation for the object type. Access to objects in a collection would also depend on the availability of indexes on that collection. The structure of objects leads to the question of what can be indexed, and the related problem of maintaining indexes [Zdo89]. For example, a property of an object could be implemented as a function, so an index can not be based on storage values.

Applying a cost measure is complicated by the need to respect encapsulation of the objects in the database. A query and the collection over which it ranges are database objects which will act upon each other when the query is executed. The cost of a query will be the cost of that interaction, and thus involves knowledge about the query and the collection being queried. Since query objects are ad hoc, our current thought is to attach a cost evaluator to type collection (i.e. to the type object for type collection). When the evaluator is activated the query object would have to cooperate with the collection type object to allow that object to evaluate the cost of the query. In addition, the type object needs access to information (selectivity, index availability and use, etc.) about the actual collection object being queried. Thus we need to determine not only a method for evaluation, but the requirements for defining types so that information needed by the query evaluator will be accessible to the evaluator within the constraints imposed by the encapsulation of the objects needed for the evaluation.

We expect that very few query transformations will be independent of the data being accessed. Thus the determination of a cost measure and application of that measure, within the encapsulation constraints imposed by abstract data types, will be of primary importance in defining query optimization in object-oriented databases.

References

[Ban87] Francois Bancilhon et al. FAD, a Powerful and Simple Database Language. In *Proceedings of the 13th VLDB Conference*, pages 97–105, 1987.

[Ban88] Jay Banerjee, Won Kim, and Kyung-Chang Kim. Queries in Object-Oriented Databases. In *Proceedings 4th Intl. Conf. on Data Engineering*, pages 31–38. IEEE, Feb 1988.

[Car88] Michael J. Carey, David J. DeWitt, and Scott L. Vandenberg. A Data Model and Query Language for EXODUS. In *SIGMOD Proceedings*, pages 413–423. ACM, June 1988.

[Gra87] Goetz Graefe. *Rule-Based Query Optimization in Extensible Database Systems.* PhD thesis, Univ. of Wisconsin-Madison, November 1987.

[Mai87] David Maier and Jacob Stein. Development and Implementation of an Object-Oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. MIT Press, Cambridge, MA, 1987.

[Osb88] S. L. Osborn. Identity, Equality and Query Optimization. In *Advances in Object-Oriented Database Systems*, pages 346–351. 2nd International Workshop on Object-Oriented Database Systems, September 1988.

[Sha89a] Gail M. Shaw and Stanley B. Zdonik. An Object-Oriented Query Algebra. To appear in *Proceedings of the 2d Intl. Workshop on Database Programming Languages*, 1989.

[Sha89b] Gail M. Shaw and Stanley B. Zdonik. A Query Algebra for Object-Oriented Databases. Technical Report CS-89-19, Brown University, 1989.

[Zdo86] Stanley B. Zdonik and Peter Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proceedings of the Hawaii International Conference on System Science*, January 1986.

[Zdo89] Stanley B. Zdonik. Query Optimization in Object-Oriented Database Systems. In *Proceedings of the Hawaii International Conference on System Science*, January 1989.

Practical Complex Object Algebras

Scott Vandenberg

(electronic mail: scottv@cs.wisc.edu)

University of Wisconsin-Madison

1. Introduction

One of the main reasons for designing algebras for data models is so that they can be used as vehicles for query optimization in systems with non-procedural interfaces. This is especially clear in the relational model [Codd70]. Recently, a large number of algebras have been proposed for data models more powerful than the relational model (e.g., nested relations, entity-relationship models, complex object models, etc.). Some of these proposals have been highly theoretical in nature, while others have been intended for use in a specific system. This situation leads to two interesting (and related) questions which are receiving our attention.

First, this recent proliferation of data models, exemplified both by the models themselves (semantic models, object-oriented models, nested relations, complex objects with and without object identity, etc.) and by extensible database systems such as EXODUS [Care86a] and Genesis [Bato87] (which assume that no single data model will solve all data modelling problems), leads one to ponder the question of whether or not there can (or even should) be a complex object analogue of Codd's relational algebra. That is, is such a standard possible or desirable in the case of complex objects? Many of these proposed data models operate on structures which are very similar, if not identical, to each other. Should they be manifestations of a single algebra in the same way that ORACLE and INGRES, for example, are approximations of the pure relational model? Or should the underlying formal algebras be substantially different from each other?

Second, what do the characteristics of the formal algebras imply for their system-dependent counterparts? For example, what effect will reducing the power of a formal algebra by eliminating the powerset operator have on the usefulness, efficiency, and ease of implementation of an algebraic optimizer for a specific system supporting complex objects? Answering such questions can give us valuable hints on how to design an algebraic optimizer for a specific data model. This discussion is intended to identify some of the more important algebraic qualities and their implications, not necessarily to point out the correct design choices (which will vary with the model, DBMS, and application).

In the following section I will briefly discuss complex objects. Section 3 contains some comments on algebraic standards. Section 4 summarizes some conclusions drawn from a survey of recent algebras in order to clarify the second question posed above and contains an initial set of ideas on how one might evaluate the suitability of a particular algebra for a particular DBMS. A sketch of our current and planned work is given in Section 5.

2. Complex Objects

Generally, a complex object is regarded as being composed from scalar values using the set and tuple type constructors, which can be applied to scalars as well as to previously formed sets and tuples. Some proposals also include arrays as a type constructor with the same status as the set and tuple constructors. Aside from its structure, a complex object may or may not have "object identity"; this depends on whose definition of "complex object" one uses. At any rate, it is an issue orthogonal to structure. An object is said to have its own identity if it can be accessed via an unchanging identifier or surrogate. Objects without identity can only be referenced by specifying one or more of the scalar values which make up the object. The remainder of this discussion will concern algebras for complex objects and for models which are strictly less powerful (in terms of modelling capabilities) than complex objects. For example, the relational and nested relational models are subsumed by the complex object model: relations are sets of tuples in which the tuples contain only scalar values and nested relations are sets of tuples in which the tuples

contain either scalar values or other relations (sets of tuples).

3. What About a Standard?

An important question is whether or not we need or want an algebra general or powerful enough to be called "the" complex object algebra. As mentioned in the introduction, the variety of recently proposed models is staggering and it is not at all clear that a single model will emerge as the most popular one like the relational model did in the 1970s. Both researchers and end-users have become aware of this variety, and it seems unlikely that anyone will want to substitute a single model for the wide range of choices now available (unless, of course, somebody comes up with the perfect data model). This is true not only because of the different (and loose) classifications of models as object-oriented, semantic, etc., but also because within each of these classifications exists a large number of data models with differences substantial enough to require a different algebra. For example, some complex object models want to support arrays as primitive type constructors and others do not; some require objects to have identity, some do not permit it, and others make it optional. An algebra sufficient for all such complex object models would have several "useless" operators in any model which did not support all of the constructs for which the algebra was designed.

With this in mind, it seems that what is needed is not a single algebra but a method and some guidelines for designing algebras for the various complex object (and other) models currently under investigation. Of course this would mean missing out on the benefits of having a standard algebra (a common foundation for research and comparison, for example), but these benefits might be at least partially regained if there are some generally agreed upon methods and guidelines for algebra design, implementation, and analysis. The next section expands upon this point.

4. Some Algebraic Characteristics and Their Consequences

After surveying 28 algebras more powerful than the relational algebra, what became apparent was the central nature of the relational algebra--virtually all of the algebras are based on this algebra in some form or another. The most important common theme identified was the inadequacy of the relational model and its algebra. Specifically, the algebraic extensions employed in the more sophisticated models imply that the relational model falls short in the areas of modelling constructs (not rich enough), expressive power (can not do, e.g., a general transitive closure), computational power (absence of aggregates, etc.), historical databases (i.e., we can not express the query "what was true at time x?"), and object identity notions (which can help alleviate some of the anomalies encountered in first normal form relations).

One motivation for the survey was to examine the operators defined for post-relational algebras and to identify the ways in which they extend or supplant the relational operators. The most popular sets of primitives seem to be based on the relational algebra, and two fairly standard sets of operators seem to be catching hold in the area of nested relations (specifically, one set for nested relations which conform to a normal form known as "Partitioned Normal Form" and one set for general nested relations). But for data more complex than nested relations, a clearly favorite set of primitives has not yet emerged (indeed, only a handful of algebras for complex objects exist). More work remains to be done in this area, but we can identify several important design considerations at this point.

One of these is the amenability of the operators to standard algebraic optimization techniques. For example, a small set of extremely powerful operators may allow fewer alternatives for query processing since it provides only a small number of algebraic transformation rules, while a large set might provide too many alternatives to consider in a reasonable amount of time. The nature and efficiency of the optimization process will probably be very sensitive to the addition/removal of new operators and it is conceivable that additional changes to the algebra may be necessary to ensure that an optimizer based upon it will be effective. And if, as discussed in the previous section, we have an algebra that is more powerful than the system employing it, the set of transformation rules may be too large in the sense that some of them will never be used and too small in the sense that the more restrictive domain of structures available in the model may allow for a larger set of possible transformations. What we seek, then, is a fairly tight fit between an algebra and a particular DBMS--the algebra should model exactly the set of queries expressible in the end-user language (which is frequently a calculus-based language).

That, of course, would be the ideal situation. In reality there rarely seems to be such a tight fit. For example, most commercial relational DBMSs support aggregate functions, which have no place in the

relational algebra. This raises the issue of what to do with query language constructs that may be difficult to model in an algebra; i.e., what should the *scope* of the algebra be--should it model the entire query language or not? The relational algebra can be extended to include aggregates, but if arbitrarily complex aggregates are allowed, what should be the legal algebraic transformations? In [Klug82], aggregates are added to the relational algebra, but the algebra has no knowledge of the semantics of a particular aggregate, making it much more difficult to come up with transformation rules involving aggregate usage. In order to more effectively manipulate expressions with aggregates, the algebra would need some knowledge of what the aggregate does. Unfortunately this would often involve giving the algebra the ability to actually compute aggregates, which would significantly increase its power [Chan81, Chan88]. Such an increase in power brings the algebra one step closer to a full programming language, making optimization much more difficult (e.g., there are many more possible transformation rules, and such rules would need to be defined for every such aggregate function). The issue for a particular algebra, then, is how much power can be added to it while preserving its utility and tractability.

Another (simpler) example of an algebraic design decision for a complex object algebra is whether or not to include the powerset operator in the algebra. Inclusion of this operator gives some of the algebras proposed in the literature the same power as most complex object calculi, while omission of the operator renders the algebra strictly less powerful than the calculus (for example, recursive queries can no longer be expressed) [Hull87]. Unfortunately, the powerset operator is inherently exponential, thus one does not want to include it in the algebra unless it is truly needed to answer certain classes of queries.

There are many other design issues to consider. For example, is the usual set-based query processing paradigm still the best choice for complex object algebras? That is, is there a viable alternative to treating all inputs and outputs of algebraic expressions as sets in light of the fact that type constructors may be applied in any order? This will have an effect on the nature of the available operators.

A "good" algebra should bridge the gap between theory and practice by taking these and other issues into account. Aside from the speed and effectiveness of query optimization mentioned above, an algebra designer should also consider the ease of implementation of the operators, the ease with which cost and other estimation functions can be specified for the operators, and whether or not the algebra is intended as an end-user or as an intermediate language.

This discussion is not meant to exhaust either the important theoretical issues or their (equally important) practical consequences.

5. Our Work

As a test vehicle for the EXODUS extensible database system being developed here at UW-Madison, we are implementing of a complex object model (the EXTRA/EXCESS model, see [Care88a]) with sets, arrays, and tuples as the available type constructors. Types may be defined using these constructors in any order, and an "object identity indicator" may be placed before any of the constructors wherever they appear (or before a scalar) to signify that the following (sub)structure of each object of the type being defined has its own object identity. An algebra design taking into account the above ideas is complete, and an implementation of the optimizer using the EXODUS optimizer generator [Grae87] will begin shortly, also with the above ideas in mind.

We also plan to quantify some of the more subjective characteristics and consequences described in Section 4 by varying the characteristics of the algebra accordingly.

BIBLIOGRAPHY

[Abit86] S. Abiteboul and N. Bidoit, "Non First Normal Form Relations: An Algebra Allowing Data Restructuring," *J. Computer and System Sciences* 33, 1986.

[Abit88a] S. Abiteboul and C. Beeri, "On the Power of Languages for the Manipulation of Complex Objects," Technical Report No. 846, INRIA, May 1988.

[Abit88b] S. Abiteboul and R. Hull, "Update Propagation in a Formal Semantic Model," *Data Eng.* 11(2), June 1988.

[Aho79] A. Aho and J. Ullman, "Universality of Data Retrieval Languages", *Proc. Conf. on Principles of Programming Languages*, 1979.

[Aris83] H. Arisawa, K. Moriya, and T. Miura, "Operations and the Properties on Non-First-Normal-Form Relational Databases," *Proc. VLDB Conf.*, Florence, Italy, October, 1983.

[Banc87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, a Powerful and Simple Database Language," *Proc. VLDB Conf.*, Brighton, England, 1987.

[Banc88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez, "The Design and Implementation of O_2 an Object-Oriented Database System," Tech. Report 20-88, Altair, April 1988.

[Banc86] F. Bancilhon and S. Khoshafian, "A Calculus for Complex Objects," *Proc. PODS Conf.*, Cambridge, MA, March 1986.

[Bane87a] J. Banerjee, H.-T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H.-J. Kim, "Data Model Issues for Object-Oriented Applications," *ACM Trans. Office Info. Sys.* 5(1), Jan. 1987.

[Bane87b] J. Banerjee, W. Kim, and K.-C. Kim, "Queries in Object-Oriented Databases," Tech. Report DB-188-87, MCC, Austin, Texas, June 1987.

[Bato87] D. Batory, "Principles of Database Management System Extensibility," *Database Eng.*, June 1987.

[Bato84] D. Batory and A. P. Buchmann, "Molecular Objects, Abstract Data Types, and Data Models: A Framework," *Proc. Tenth VLDB Conf.*, Singapore, Aug. 1984.

[Brac79] R. J. Brachman, "On the Epistemological Status of Semantic Networks," in *Associative Networks*, N. V. Findler (Ed.), Academic Press, New York, 1979.

[Care86a] M. Carey, D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. Richardson, and E. Shekita, "The Architecture of the EXODUS Extensible DBMS," *Proc. Int'l. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986.

[Care86b] M. Carey and D. DeWitt, "Extensible Database Systems", *Proc. Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1986.

[Care88a] M. Carey, D. DeWitt, and S. Vandenberg, "A Data Model and Query Language for EXODUS," *Proc. SIGMOD Conf.*, Chicago, Illinois, 1988.

[Care88b] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview", Comp. Sci. Tech. Report #808, Univ. of Wisconsin, Madison, Wisconsin, November, 1988.

[Ceri87] S. Ceri, S. Crespi-Reghizzi, L. Lavazza, and R. Zicari, "ALGRES: A System for the Specification and Prototyping of Complex Databases," Tech. Report 87-018, Dipartimento di Elettronica, Politecnico di Milano, 1987.

[Chan81] A. Chandra, "Programming Primitives for Database Languages", *Proc. Conf. on Principles of Programming Languages*, 1981, pp. 50-62.

[Chan88] A. Chandra, "Theory of Database Queries", *Proc. Conf. on Principles of Database Systems*, 1988, pp. 1-9.

[Chen76] P. Chen, "The Entity-Relationship Model — Toward a Unified View of Data," *ACM Trans. Database Sys.* 1(1), March 1976.

[Codd70] E. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. ACM* 13(6), June 1970.

[Codd79] E. Codd, "Extending the Relational Model to Capture More Meaning," *ACM Trans. Database Sys.* 4(4), Dec. 1979.

[Codd80] E. Codd, "Data Models in Database Management," *Proc. Workshop on Data Abstraction, Databases, and Conceptual Modeling"*, Pingree Park, Colorado, 1980.

[Dada86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch, "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View of Flat Tables and Hierarchies," *Proc. SIGMOD Conf.*, Washington, DC, 1986.

[Desh88] A. Deshpande and D. Van Gucht, "An Implementation for Nested Relational Databases," *Proc. VLDB Conf.*, Los Angeles, CA, 1988.

[Desh87] V. Deshpande and P.-A. Larson, "An Algebra for Nested Relations," Research Report CS-87-65, University of Waterloo, Dec. 1987.

[Fisc83] P. C. Fischer and S. J. Thomas, "Operators for Non-First-Normal-Form Relations," *Proc. IEEE COMPSAC*, 1983.

[Fish87] D. Fishman, D. Beech, H. Cate, E. Chow, T. Connors, J. Davis, N. Derrett, C. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. Neimat, T. Ryan, and M. Shan, "Iris: An Object-Oriented Database Management System," *ACM Trans. Office Info. Sys.* 5(1), Jan. 1987.

[Grae87] G. Graefe and D. DeWitt, "The EXODUS Optimizer Generator," *Proc. SIGMOD Conf.*, San Francisco, CA, May 1987.

[Grae88] G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: The REVELATION Project", Tech. Report CS/E 88-025, Dept. of Computer Science and Engineering, Oregon Graduate Center, 1988.

[Guck88] R. L. Guck, B. Fritchman, J. Thompson, and D. Tolbert, "SIM: Implementation of a Database Management System Based on a Semantic Data Model," *Data Eng.* 11(2), June 1988.

[Guti87] R. Guting, R. Zicari, and D. Choy, "An Algebra for Structured Office Documents", IBM Research Report RJ 5559 (56648), IBM Almaden Research Center, March 1987.

[Gyss88] M. Gyssens and D. Van Gucht, "The Powerset Algebra as a Result of Adding Programming Constructs to the Nested Relational Algebra," *Proc. SIGMOD Conf.*, Chicago, Illinois, June 1988.

[Hall84] P. A. V. Hall, "Relational Algebras, Logic, and Functional Programming," *Proc. SIGMOD Conf.*, Boston, Massachusetts, 1984.

[Hamm81] M. Hammer and D. McLeod, "Database Description with SDM: A Semantic Database Model", *ACM TODS* 6(3), September 1981.

[Horn87] M. Hornick and S. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM Trans. Office Info. Sys.* 5(1), Jan. 1987.

[Houb87] G. J. Houben and J. Paredaens, "The R^2-Algebra: An Extension of an Algebra for Nested Relations," Tech. Report 87/20, Dept. of Math. and Computing Sci., Computing Sci. Section, Eindhoven Univ. of Tech., December 1987.

[Houb88] G. J. Houben, J. Paredaens, and D. Tahon, "Expressing Structured Information using the Nested Relational Algebra: An Overview," *Proc. Eighth SCCC Int. Conf. Comp. Sci.*, Santiago, July 1988.

[Hull87] R. Hull, "A Survey of Theoretical Research on Typed Complex Database Objects", in *Databases*, ed. J. Paredaens, Academic Press, London, 1987.

[Hull88a] R. Hull and J. Su, "Untyped Sets, Invention, and Computable Queries," extended abstract, submitted to *ACM Symp. Principles of Database Sys.*, March 1989.

[Hull88b] R. Hull, "Four Views of Complex Objects: A Sophisticate's Introduction", draft, Dept. of Computer Science, Univ. of Southern California, Los Angeles, California, May 1988.

[Jaes82a] G. Jaeschke, "An Algebra of Power Set Type Relations," TR 82.12.002, IBM Heidelberg Scientific Center, Dec. 1982.

[Jaes85a] G. Jaeschke, "Recursive Algebra for Relations with Relation Valued Attributes," TR 85.03.002, IBM Heidelberg Scientific Center, March 1985.

[Jaes85b] G. Jaeschke, "Nonrecursive Algebra for Relations with Relation Valued Attributes," TR 85.03.001, IBM Heidelberg Scientific Center, March 1985.

[Jaes82b] G. Jaeschke and H.-J. Schek, "Remarks on the Algebra of Non First Normal Form Relations," *Proc. ACM PODS Conf.*, Los Angeles, CA, 1982.

[Jark84] M. Jarke and J. Koch, "Query Optimization in Database Systems," *Comp. Surveys* 16(2), June 1984.

[Jhin88] A. Jhingran, "A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedural Objects," *Proc. VLDB Conf.*, Los Angeles, California, 1988.

[Kent79] W. Kent, "Limitations of Record-Based Information Models," *ACM Trans. Database Sys.* 4(1), March 1979.

[Khos87] S. Khoshafian and P. Valduriez, "Sharing, Persistence, and Object Orientation: A Database Perspective," DB-106-87, MCC, April 1987.

[Klug82] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *J. ACM* 29(3), July 1982.

[Kort88] H. Korth, "Optimization of Object-Retrieval Queries (extended abstract)," Dept. of Computer Sciences, Univ. of Texas, Austin, Texas, April 1988.

[Kupe85] G. M. Kuper, "The Logical Data Model: A New Approach to Database Logic," PhD. Thesis, Dept. of Computer Science, Stanford University, Stanford, CA, Sept. 1985.

[Lecl87] C. Lecluse, P. Richard, and F. Velez, "O_2, an Object-Oriented Data Model," *Proc. SIGMOD Conf.*, Chicago, IL, 1988.

[Maie86a] D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," *Proc. 1st OOPSLA Conf.*, Portland, OR, 1986.

[Maie86b] D. Maier and J. Stein, "Indexing in an Object-Oriented DBMS," Tech. Report CS/E-86-006, Oregon Grad. Center, Beaverton, Oregon, May 1986.

[Maie86c] D. Maier, "A Logic for Objects," Tech. Report CS/E-86-012, Oregon Grad. Center, Beaverton, Oregon, Nov. 1986.

[Mano86] F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model," *Proc. Int'l. Workshop on Object-Oriented Database Sys.*, Asilomar, CA, Sept. 1986.

[Mylo80] J. Mylopoulos, P. Bernstein, and H. Wong, "A Language Facility for Designing Database-Intensive Applications", *ACM TODS* 5(2), June 1980.

[Ozso87] G. Ozsoyoglu, Z. Ozsoyoglu, and V. Matos, "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions," *ACM Trans. Database Sys.* 12(4), Dec. 1987.

[Ozso83] Z. Ozsoyoglu and M. Ozsoyoglu, "An Extension of Relational Algebra for Summary Tables," *Proc. 2nd Int. Workshop Stat. Database Mgmt.*, Lawrence Berkeley Labs., Univ. of California, Berkeley, 1983.

[Pare88] J. Paredaens and D. Van Gucht, "Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions," *Proc. ACM PODS Conf.*, 1988.

[Pare85] C. Parent and S. Spaccapietra, "An Algebra for a General Entity-Relationship Model," *IEEE Trans. Software Eng.* 11(7), July 1985.

[Pist86] P. Pistor and F. Andersen, "Designing a Generalized NF2 Model with an SQL-Type Language Interface," *Proc. Twelfth VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[Roth87] M. Roth, H. Korth, and D. Batory, "SQL/NF: A Query Language for ¬1NF Relational Databases," *Inform. Systems* 12(1), 1987.

[Roth88] M. Roth, H. Korth, and A. Silberschatz, "Extended Algebra and Calculus for ¬1NF Relational Databases," *ACM Trans. Database Sys.*, 13(4), December 1988.

[Sche85] H.-J. Schek, "Towards a Basic Relational NF^2 Algebra Processor," *Proc. Int. Conf. on FODO, Kyoto, Japan, 1985.*

[Sche88] H.-J. Schek, "Nested Relations, a Step Forward or Backward?," *Data Eng.* 11(3), September 1988.

[Sche86] H.-J. Schek and M. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Sys.* 11(2), 1986.

[Scho86] M. H. Scholl, "Theoretical Foundations of Algebraic Optimization Utilizing Unnormalized Relations," *Proc. Int. Conf. Database Theory*, Rome, 1986.

[Scho87a] M. H. Scholl, H.-B. Paul, and H.-J. Schek, "Supporting Flat Relations by a Nested Relational Kernel," *Proc. VLDB Conf*, Brighton, England, Sept. 1987.

[Scho87b] M. H. Scholl and H.-J. Schek, eds., *Theory and Applications of Nested Relations and Complex Objects: An International Workshop*, Darmstadt, Germany, April 1987.

[Ston86] M. Stonebraker, "Inclusion of New Types in Relational Database Systems," *Proc. 2nd Data Eng. Conf.*, Los Angeles, CA, Feb. 1986.

[Tsur86] S. Tsur and C. Zaniolo, "LDL: A Logic-Based Data Language", *Proc. 12th VLDB Conf.*, Kyoto, Japan, August 1986.

[Vald86] P. Valduriez, S. Khoshafian, and G. Copeland, "Implementation Techniques of Complex Objects," *Proc. Twelfth VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[VanG87] D. Van Gucht, "On the Expressive Power of the Extended Relational Algebra for the Unnormalized Relational Model," *Proc. ACM PODS Conf.*, 1987.

[VanG86] D. Van Gucht and P. Fischer, "Some Classes of Multilevel Relational Structures," *Proc. ACM PODS Conf.*, 1986.

[Zani83] C. Zaniolo, "The Database Language GEM," *Proc. SIGMOD Conf.*, San Jose, CA, 1983.

[Zill80a] S. N. Zilles, "An Introduction to Data Algebras," *Copenhagen Winter School Proc.*, LNCS #86, 1980.

[Zill80b] S. N. Zilles, "Types, Algebras, and Modeling," *Proc. Workshop on Data Abstraction, Databases, and Conceptual Modeling"*, Pingree Park, Colorado, 1980.

Using Object-Oriented Subtyping in Query Optimization and Processing

J. Eliot B. Moss

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003
Moss@cs.umass.edu; (413) 545-4206

An outgrowth of the Mneme persistent object store project at UMass [Moss and Sinofsky, 1988] has been the realization that while a store can provide more appropriate semantics (e.g., object identity) for object-oriented databases, database programming languages, and persistent programming languages, being able to understand, manipulate, and process *code*, specifically queries, as well as data objects is crucial to providing adequate performance for data-intensive object-oriented applications. This point is driven home by considering object servers in a distributed client-server model. Suppose a query to be evaluated is rather selective but (because there happens to be no relevant index) involves a scan through a considerable number of objects. Given an object store approach, *all* the objects must be brought to the client, since the object store provides only create/delete, read/write operations. If the objects could be filtered before being brought to the client, somewhat like the System R RSS subsystem, considerable network traffic and overhead would be saved. Rather than building in particular features, such as "scan a set under a predicate" (probably drawn from a restricted class of predicates), the extensibility and open-endedness of object-oriented systems suggest a more general model of execution in which processed (optimized) code can be divided between the client and the server. This is somewhat analogous to traditional distributed database query processing—we expect every node to be able to interpret query plans and/or execute compiled queries.

Our particular interest is in bridging the gap between programming languages and database query languages, so we will attempt to optimize programs written in a language that is fully general (i.e., Turing complete), as opposed to a restricted query language. To have much hope of doing effective optimization, we must choose the built-in data types, and the constructors for new types, for such a language very carefully. While we do not yet have a complete answer to that problem, in this position paper we sketch some ideas and try to articulate the direction we are heading.

1 Sets and relationships

While a fully general object-oriented language will certainly allow the construction of arbitrary new abstract data types, we focus on two kinds of types that are most relevant to query optimization: *sets* and *relationships*. Sets provide the means to aggregate similar

Using Object-Oriented Subtyping in Query Optimization and Processing

J. Eliot B. Moss

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003
Moss@cs.umass.edu; (413) 545-4206

An outgrowth of the Mneme persistent object store project at UMass [Moss and Sinofsky, 1988] has been the realization that while a store can provide more appropriate semantics (e.g., object identity) for object-oriented databases, database programming languages, and persistent programming languages, being able to understand, manipulate, and process *code*, specifically queries, as well as data objects is crucial to providing adequate performance for data-intensive object-oriented applications. This point is driven home by considering object servers in a distributed client-server model. Suppose a query to be evaluated is rather selective but (because there happens to be no relevant index) involves a scan through a considerable number of objects. Given an object store approach, *all* the objects must be brought to the client, since the object store provides only create/delete, read/write operations. If the objects could be filtered before being brought to the client, somewhat like the System R RSS subsystem, considerable network traffic and overhead would be saved. Rather than building in particular features, such as "scan a set under a predicate" (probably drawn from a restricted class of predicates), the extensibility and open-endedness of object-oriented systems suggest a more general model of execution in which processed (optimized) code can be divided between the client and the server. This is somewhat analogous to traditional distributed database query processing—we expect every node to be able to interpret query plans and/or execute compiled queries.

Our particular interest is in bridging the gap between programming languages and database query languages, so we will attempt to optimize programs written in a language that is fully general (i.e., Turing complete), as opposed to a restricted query language. To have much hope of doing effective optimization, we must choose the built-in data types, and the constructors for new types, for such a language very carefully. While we do not yet have a complete answer to that problem, in this position paper we sketch some ideas and try to articulate the direction we are heading.

1 Sets and relationships

While a fully general object-oriented language will certainly allow the construction of arbitrary new abstract data types, we focus on two kinds of types that are most relevant to query optimization: *sets* and *relationships*. Sets provide the means to aggregate similar

objects so that they can be treated in bulk, and also so that interesting members of the set can be located according to their properties rather than by their identity alone. In many cases, the point is, in fact, to *discover* the identity of the objects in a set that satisfy a property. Relationships play the role of tuples: a given relationship (which may itself be an object) relates two or more specific objects. The correlate of a relation would be a set of relationship objects. Relationship objects differ from tuples in that they are *objects*, possessing identity. They can also possess attributes, in addition to indicating the objects related, and the attributes may be mutable. Furthermore, in general one can discover from an object the relationships of a particular type in which the object participates in a given role, etc. This sketch is necessarily brief, but we should point out that there are a number of details wee have not worked out yet to our satisfaction. Still, let us continue with a discussion of subtyping.

Suppose for a moment that there is only one set type constructor, `set`, that defines a type for each element type; i.e., `set[foo]` is the type of a set that contains objects of type `foo`. Similarly, for each list of name-type pairs we define a corresponding relationship type, such as `relationship[advisor: professor, advisee: student]`, given that `professor` and `student` are object types. Thus far this model is not particularly different from the FAD model [Bancilhon *et al.*, 1987], among others. What we wish to do, though, is to impose a truly object-oriented view on this model, in which types correspond to behaviors, and behavior is exhibited by the semantics of the operations available to create and manipulate objects of a type. Thus, `set[...]` is important because we can filter (select) members of a set, insert and remove members, etc. Our view of relationship objects is that it must be possible (in general, at least) to find the relationship objects in which a given object participates in a given role (e.g., the `advising` relationship objects having a given `professor` in the `advisor` role, and to find all role members from the relationship object. In this way we could find all the students a professor advises, and so forth.

2 Behavioral subtyping

The importance of shifting to the behavioral view is that it makes clear that, as far as semantics is concerned, it does not matter how a given set or relationship is implemented—we just need to insure that each set acts like a set. So, we could have a number of implementations of sets, that provide different performance for different operations. This raises the issue of having the compiler and optimize jointly choose the appropriate representation, based on program analysis, programmer directives, run-time statistics, or any other basis for a good choice. (Our research will also consider the problem of changing these decisions later, in response to changes in the data or the decision criteria, a process we call *adaptive reoptimization* that we will not explore deeply here.) Even once a given representation has been chosen, it is well known that a given operation can still be carried out in more than

one way, e.g., a file scan or an index scan when an index is available. That decision can be made at compile-time or deferred until run-time, with dynamic query plans being an intermediate alternative. We suspect that partial evaluation of compiler/optimizer functions may be a fruitful approach to examine, but discussing that would be a digression from the thoughts we are trying to express here.

The point is that we can have a number of different implementations of sets, but all of them are subtypes of (or equivalent to) the original `set[...]` types, according the the notion of *behavioral subtyping*. A behavioral subtype is defined not as *inheriting from* its supertype but as *acting like* the supertype. That is, object of a behavioral subtype should be substitutable for object of its supertypes: they should provide all of the supertype's functionality, though they may provide more. This notion, as well as the distinction between inheritance and subtyping are explored further in [Moss and Wolf, 1988] (submitted for publication).

3 Exploiting special cases

While different implementations of general sets and the operations on them are certainly interesting, we are interested in something that goes further: taking advantage of special cases. There are a vast number of special cases of sets, and of sets of relationships in particular. A set might be ordered (or be able to supply its members in some interesting orders), it might be restricted as to the nature of its members (e.g., have a key constraint), it might restrict which objects can be added or removed, it might have limited cardinality, etc. Sets of relationships can have key, cardinality, and other constraints. Individual relationships can have semantic constraints. Some of these special properties add behavior, and some remove it. In addition to the various sort of restrictions mentioned so far, a given application may not need all the functionality offered by general sets or relationships. Since we are talking about adding and removing operations from sets and relationship types, we are really talking about a large collection of different but similar and related types. The important thing is that these different types allow for an even wider range of implementations, both in terms of representation and in terms of coding the operations. Thus, "subtypes" (some of the types may actually be supertypes) of the set type become very interesting to explore.

As an example, let us consider the relationship between wires and gates in a simple CAD model. We might represent a wire as a set of relationships between a gate output and some gate inputs, of type `relationship [driver: output, driven: input]`. This could be implemented in the traditional relational way by having a single set containing all members of this relationship. If we have a gate output and would like to know the inputs to which it is connected, we do a filter (selection) on the whole set with a predicate indicating that we are interested only in relationships having the given output in the `driver` role. A symmetrical filter allows us to locate the driver of a given input line, etc.

We can retain the conceptual appeal of this relational view of the world, yet obtain highly efficient navigational access if we use an appropriate subtype for implementing the set of relationships representing the gate connections. One important restriction we will assume is that we never need to enumerate the entire set: we are only ever interested in the two filter queries that take us from a driver to the driven components and vice versa. A second restriction we will assume is that we never need the relationship objects *as objects*—we use them only to traverse to the other member of the relationship. Making these assumptions, each output and input can be directly connected to the set of things connected to it, so that no additional filtering is required to execute the filter queries. Third, we impose the reasonable cardinality constraint that an input be connected to at most one driver. Given that knowledge, we can represent the driven-to-driver part of the relationship by a direct pointer from the input object back to the output object that drives it. Since an output may drive more than one input, we use a slot in the output objects to point to a data structure representing the set of input object connected. This data structure itself might turn out to be threaded through the input object if we like.

The point is that we envision that the compiler and optimizer can choose very efficient representations if provided the correct "subtype" of set to be implemented. If the assumptions that allowed a given subtype to be used are changed later, we simply recompile the code (and, less trivially, convert the existing objects to the new representation somehow). We are currently involved in trying to enumerate the interesting cases in the limited setting of binary relationships, as well as trying to get a handle on how to express the different cases in a reasonable and convenient notation that might form the basis for a programming language fragment.

One of the interesting questions here is whether or not we should try to *recognize* special cases from examination of the code, or whether we should require *declarations* of the interesting properties. It is not simply an issue of making things easier for the compiler, since a declaration is a cross-check that must be changed explicitly. Thus, a declaration could guard against an accidental and silent transformation of a very efficient form into a less efficient one.

4 More special cases

There are several other useful things we would like to do in this framework. First, since the object-oriented view suggests that attributes are obtained by invoking a function given an object's identifier, we should be able to group some attributes together (the traditional notion of an object in, e.g., Smalltalk) but handle others separately by storing them in a dictionary keyed by identifier. This is especially useful when a tool needs to add its own annotations to objects of existing types: the annotations can be stored separately if we like. Another thing we need to understand better is indexing based on the results of operations. We need to recognize the special case where the result is a stored attribute of the object,

and especially when it is a well known type such as integer or string. Third, we want to understand derived data and constraints much better. Finally, we note that the object-oriented approach pretty much automatically gives us extensible query processing—any new set-like type need only come with the appropriate operations, and the object-oriented language's execution system, which we assume includes dynamic binding (method lookup), will "do the right thing". Getting optimization to do useful things with the new type requires more work, and an overall framework for expressing and performing optimizations remains an interesting and challenging problem.

5 Conclusion

We have described some ways in which the notion of behavioral subtypes in an object-oriented language can be used in query optimization and query processing. First, a whole variety of closely related types, namely those defining sets, relationships, and similar behaviors, can provide a wide range of implementations of aggregations of objects, offering efficient representations tailored to their uses. Second, such a collection of types also supports flexible query execution, allowing the same optimized query code to be used for a variety of slightly different underlying representations. The query optimization process, which we see as a broader program optimization process, would choose representations (appropriate subtypes) for new or intermediate sets (and possibly suggest a change of representation to existing sets). It would also need to transform code from the offered form to a more efficient form. A combination of statically gathered information (available from program analysis or declarations) could be used in the optimizer's decision making process, in addition to static and dynamic statistics on the sets. Having a variety of slightly different set types should be a benefit, since it enables more rapid recognition of important special cases or limitations, while allowing general transformations for those operations many types have in common. We would expect, then that each new subtype added would require only a few new rules in a rule based query optimizer. The point is that we may not need vast new capabilities, only incremental extension of understood techniques.

References

[Bancilhon *et al.*, 1987] François Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. FAD: A powerful and simple database language. In *Proceedings of the 13th International Conference on Very Large Databases* (Brighton, England, Sept. 1987), Morgan Kaufmann, pp. 97–105.

[Moss and Sinofsky, 1988] J. Eliot B. Moss and Steven Sinofsky. Managing persistent data with Mneme: Designing a reliable, shared object interface. In *Advances in Object-Oriented Database Systems* (Sept. 1988), vol. 334 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 298–316.

[Moss and Wolf, 1988] J. Eliot B. Moss and Alexander L. Wolf. Toward principles of inheritance and subtyping in programming languages. COINS Technical Report 88-95, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, Nov. 1988.

Query Optimization in Object-Oriented Databases[1]

Girish Pathak, Texas Instruments Incorporated, and
José A. Blakeley, Indiana University

Summary

At Texas Instruments we have been developing a distributed Object-Oriented Database system (OODB), Zeitgeist [12], for Computer Aided Design (CAD) and Computer Integrated Manufacturing (CIM) applications. The Zeitgeist/OODB system has been in use by TI's VLSI designers since 1987. The Zeitgeist system is intended to be navigated (about 90% of the time) or accessed through the native object-oriented programming environment. We are also developing a set-oriented interface for Zeitgeist/OODB users. The set-oriented interface will be used to perform retrieve/update (about 10% of the time) from the database as the top level querying facility that will be followed by object-oriented computing. A linguistic interface of the query system is already under development and is referred to as Object Query Language (**OQL**). This paper briefly describes the research issues in OODB query optimization area, the OQL object model, and identifies some of the query optimization techniques that may prove useful for OODB systems.

Because of the large size of the object-base for CAD/CIM applications, it is clear that query optimization will be necessary to provide faster response. The research works in the area of query optimization for conventional databases like relational systems provide an initial step towards developing query optimization for our OQL system. However, most of the query optimization techniques in OODBs are likely to be different from the relational systems which enjoy a consensus over their data model and data manipulation language. In the OODB arena, although several OODB data models have been proposed [1, 3, 5], it is still not clear to us whether one can define an all comprehending data model for OODB systems. Aside from the data model issue, the OODB query systems must optimize queries over methods (i.e., computed functions). Some of these methods may have side-effects!

For the purposes of query system, OQL assumes a simple object data model of object-oriented programming languages with (encapsulated) objects, object identifiers, and type inheritance. Objects can be simple primitive objects or composite objects with references to other objects. We propose to develop an OODB query system, that draws its semantics

[1]For further information, contact Dr. Girish Pathak at Phone: (214) 995-0662; E-mail: pathak@csc.ti.com; and Address: Computer Science Center, Texas Instruments Incorporated, P. O. Box 655474 MS 238, Dallas, Texas 75265.

from the relational calculus and can be used to query over simple values, complex objects with methods, and can perform navigational queries through the class hierarchy.

Consider, for instance, an employee type, which is a specialization on person (i.e., inherits properties from type person), and has attributes name, birth-date, a reference to its picture, and a reference to a set children which are of type person. It is reasonable to have a user request to retrieve all the descendants of a particular employee with the condition that each descendant must be younger than 18 and has a picture with gray level of 10.0. Here, age is a method or computed value. One would expect such a request to be processed by the OODB system within reasonable time limit. This example reveals various components of OODB query interface: complex objects, inheritance hierarchy, recursion, transitive closure, and object-identity. The following list describes various query optimization issues in OODB systems:

- Indexing and clustering: Some of the interesting research areas in OODB query systems are indexing over attributes [9], clustering and prefetching over objects. In our Zeitgeist system, we are just beginning to obtain some preliminary results on our approaches of object prefetching and clustering.

- Caching the computation: This issue is a generalization of the problem of processing queries through views in relational and extensible systems. The methods in an OODB may be too complex to compute repeatedly. In such situations it is even more important to rely on some form of cached or precomputed methods to be able to process queries efficiently. Data caching mechanisms similar to view materialization [4, 7] and support of stored procedures in extensible database systems [11] may be applicable not only for the optimization over methods with no side-effects but also for materialized transitive closures.

- Computation at client vs. server site: This issue has more to do with query optimization in distributed database systems. The client vs. server site computation issue is important to us because Zeitgeist/OODB system has built-in features to compute methods locally at the client workstation or remotely at the server site. Our query optimization strategies will comprehend the trade-offs of local vs. remote computation to provide efficient processing.

- Optimization over method or method-combination: In an OODB system, a method or method-combination is invoked with some parameters by a user or application. An individual method invocation can fire several other methods in the process. Mathematically, one can write an expression like F1.(F2 + F3).F4 to denote that in the method-combination, F2 or F3 is invoked after method F1 and so on (the dot operator denotes ordering of method invocation whereas the plus operator represents invocation of methods with no order). If a database designer can provide some rules like

equivalence rules for method or method-combination invocation (Fn = Fa.Fb.Fc.Fd), then it would be a matter of using appropriate algebraic rules and cost functions to select an optimum method-combination from available choices. This approach is similar to what the relational model provides by using the properties of the algebra (e.g., commutativity, distributivity) in the optimization of relational expressions.

- Local methods vs. global methods: Methods may be local to objects or inherited from higher-level system objects (e.g., relation, set, collection). This distinction has some implications in the design of a set-oriented query facility in an OODB system. If the method is local, then it may be beneficial to provide a mechanism to make the implementation of methods visible to the query optimizer. This is the issue addressed by Graefe and Maier in the Revelation project [6]. If the method is global, that is, it is available through a system-based object, then it may be possible to encapsulate a generic query processing strategy to compute, for example, selects or projects on objects [10].

As discussed earlier, object-oriented database systems are bringing new parameters to the database query optimization arena. Through OQL query system, we hope to address the aforementioned issues which are relevant to CAD/CIM applications.

References

[1] Rakesh Agrawal and N. H. Gehani. "ODE (Object database and environment): The language and the data model," *In Proceedings of ACM SIGMOD International Conference on Management of Data*, 1989.

[2] Francois Bancilhon. "Object-Oriented Database Systems." *In Proceedings of 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*," 1988.

[3] Jay Banerjee, Won Kim, and Kyung-Chang Kim. "Queries in Object-Oriented Databases", *In Proceedings of Fourth International Conference on Data Engineering*, Los Angeles, pp. 31–38, California, February, 1988.

[4] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. "Efficiently updating materialized views," *In Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 61-71, Washington, DC, May 1986.

[5] Carey, D. J. DeWitt and S. L. Vandenberg. "A data model and query language for EXODUS," *In Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 413–422, September 1988.

[6] Goetz Graefe and David Maier. "Query optimization in Object-Oriented Database Systems: A prospectus," Lecture Notes in Computer Science, Advances in Object-Oriented Database Systems, *2nd International Workshop on Object-Oriented Database Systems*, pp. 359–363, Published by Springer-Verlag, September 1988.

[7] Eric Hanson. "A performance analysis of view materialization strategies," *In Proceedings of ACM SIGMOD International Conference on Management of Data*, 1987.

[8] David Maier. "Why isn't there an object-oriented data model," *To appear in Proceedings of IFIP*, 1989.

[9] David Maier and Jacob Stein. "Indexing in an object-oriented DBMS," *In Proceedings of International Workshop on Object-Oriented Database Systems*, pp. 171–182, Pacific Grove, California, September 1986.

[10] S. L. Osborn. "Identity, equality and query optimization," Lecture Notes in Computer Science, Advances in Object-Oriented Database Systems, *2nd International Workshop on Object-Oriented Database Systems*, pp. 359–363, Published by Springer-Verlag, September 1988. 1988.

[11] Michael Stonebraker, J. Anton, and E. Hanson. "Extending a Database System with Procedures." *ACM Transactions on Database Systems*, Vol. 12, No. 3, pp. 350–376, September 1987.

[12] S. Ford, J. Joseph, D. Langworthy, D. Lively, G. Pathak, E. Perez, R. Peterson, D. Sparacin, S. Thatte, D. Wells, and S. Agarwal. "ZEITGEIST: Database support for object-oriented programming," Lecture Notes in Computer Science, Advances in Object-Oriented Database Systems, *Second International Workshop on Object-Oriented Database Systems*, pp. 23–42, Published by Springer-Verlag, September 1988.

On the Complexity of Nested Relational Operations

Marc H. Scholl

Institute of Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
e-mail: scholl@inf.ethz.ch

1 Introduction

Designing and implementing a query optimizer and query processor for a nested relational database system (DBMS) requires a number of steps like deciding on operations supported on nested relations (i.e. the semantics of a query language (QL)), defining a syntax (i.e. a concrete QL for the system), investigating equivalences among expressions of the QL, devising optimization strategies or rules, and finding efficient evaluation techniques for single operations of the QL and combinations thereof. While these activities are rather theoretical in nature, they obviously have tremendous impacts on the architecture and implementation of the corresponding DBMS. We have presented powerful operations on nested relations in an algebraic syntax in [SS86], investigated equivalence rules in this algebra in [Sch86, SPS87], and described the implementation of an efficient query evaluation algorithm in [DPS86, PSS+87]. Here we focus on the following problem: typical query evaluation techniques distinguish between "simple" and "complex" operations in the QL (like selections and projections vs. joins or sorting in the relational setting). The simple operations are executed "on the fly" in a rather low level of the DBMS architecture while more complex ones are processed on top of that layer. Classical examples of such architecures are System R (RDS on top of RSS [ABC+76, CAB+81]), Ingres (MVQP on top of OVQP [SWKH76]), LOLEPOPs in Starburst are a more recent example of simple operations [Loh88].

The idea behind such a dichotomy in query processing strategies is that (combinations of) simple operations can be computed while scanning data, hence "on the fly", we will call them "*single scan operations*" in the sequel. Intuitively, we read in blocks from the disk, inspect tuples sequentially and compute the result of such operations without touching data more than once. Formally, these operations have linear complexity in the size of the involved relations. Other operations require inspecting single data elements more than once (like sorting, or joins with nested loops). The exact classification of operations into these two classes is crucial as to avoid unnecessary scans of the input data, particularly as this usually means reading in data from secondary storage.

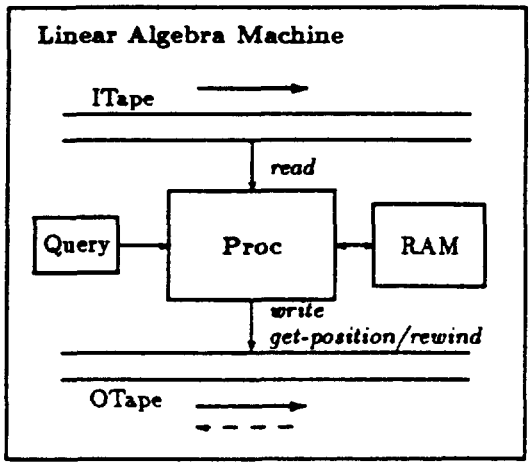
Among the relational operations, clearly selection is a single scan operation: while scanning tuples of the input relation sequentially, we can test the selection condition on each single tuple and decide whether it belongs to the result or not. Projections are only linear if we ignore duplicate elimination. Obviously, joins are not linear in general. Therefore, we find selections and projections in the lower level query processors (RSS, OVQP) and joins (together with the other non-linear operations) in the higher level query processors (RDS, MVQP) in typical relational DBMS architectures. In the flat relational context, the distinction seems to be quite evident. However, when switching to nested relations with corresponding new styles of operations things need a little more elaboration. Selection conditions may involve set comparisons, projections and selections may involve complex operations (like joins) on deeper levels of the nesting hierarchy[1]. New operations are added: nesting and unnesting. Deciding whether such operations are single scan processible or not is non-trivial in some cases.

We present a model of query processing that explicates our intuition about single scan operations. Using this model we analyse the operations individually and consider *expressions*, i.e. compositions of operations. In designing such a model we have to be very careful, as can be seen from the following simple example. Our intuitive understanding is, of course, that join is no single scan operation, however, consider the following algorithm:

```
begin
for each r ∈ R do
   read(r) into Temp
for each s ∈ S do
   { add Temp⋈s to the result }
end.
```

Obviously, the algorithm computes the correct result without "touching" (i.e. reading in) stored tuples more than once. However, we used intermediate storage space in this example, and accessed it multiply. So, what is the point? Should we prohibit any intermediate storage? The answer is negative: we will need *some* additional storage. Consider a nested relational selection (on the famous Departments and Employees

[1] We use our terminology of a nested relational algebra [SS86]. However, all investigations and results apply for other QLs too, provided they support the types of operations we consider.



(a) LAM-Architecture

```
while not { End of input }
do read (d(d#, dn, dl))
   temp := get-position (OTape)
   write (d(d#, dn, dl))
   found := false
   while not { End of subrelation }
   do read (e(e#, en, s))
      if e(en) = 'Smith'
        then found := true
      fi
      write (e(e#, en, s))
   od
   if not found
      then rewind (OTape,temp)
   fi
od
mark_end-of-tape (OTape)
```

(b) A sample program

Figure 1: The abstract linear algebra machine

–nested– relation) asking for departments that have an employee named 'Smith'. The employees working for a particular department are stored as a 'subrelation'. Hence, assuming a hierarchical pre-order scan of nested tuples we have to somewhere store department information (e.g. name and location) until we finished the (subordinate scan) of all employees. Only then can we decide whether the department matches the selection condition. So, as we want to consider such a nested selection a single scan operation, we in fact need some intermediate storage. Thus, to avoid the above problem with the join operation, we have to limit the size of additional storage used in an algorithm in order to consider it single scan.

We present our abstract model of a nested relational query processing engine for linear operations in section 2 and give our results concerning linearity of operations in section 3.

2 An Abstract Algebra Machine

Input data may be read only once, thus we use an *input (Turing) tape* that is automatically moved forward while reading data into the processor's memory. Tuples of the input (nested) relation[2] are stored in a pre-order linearization on the input tape. The units of transfer between the tape and the processor memory are so-called "atomic fragments", i.e. all atomic values of a single level of the nesting hierarchy are read in one step. E.g. in our department and employees example, there would be one fragment for each department followed by the fragments describing the individual employees (of this department), followed by the next department and so on[3]. Data is read in and used for computing results by the *processor*, which eventually writes result tuples to an *output (Turing) tape*. The ouput tape cannot be read, however, we allow rewinding the output tape back to a position that has formerly been recorded in the processors memory. This allows for dropping (department) tuples that have been considered candidate matches to a query but finally do not qualify, for instance (see the nested selection example above). Once rewinded, the content of the output tape behind the new position is lost, hence the output tape behaves like a stack with "push" and "pop" but no "top" operation.

The *processor* can issue "read" commands to the input tape, "write", "get position", and "rewind" operations to the output tape, and checks conditions on the input. It gets the query to perform on the input in some convenient form, e.g. as an operator tree. Obviously, we need at least enough memory to hold the query tree and the data units transfered by one read operation on the input tape. The sizes of these are determined by the query or the schema[4], it is important to notice that the size is independent from the extension of the database! We already mentioned that we spend some more memory to 'remember' output tape positions, the state of subconditions in complex selections and some other information. Again, we do not allow the size of that additional memory to depend on the *extension* of the relation (it may well depend on its *intention*, i.e. schema). The algebra machine can be thought of as shown in figure 1(a), where we included the set of operations valid on the two tapes. Our criterion for an operation to be single scan processible is

[2] only monadic operators will be linear, see below

[3] this exactly matches the implementation in DASDBS[DGW85, DPS86]

[4] the maximum lengths of all atomic attributes of the fragments, an information that is usually found in the DB catalogs

Definition 1 *An operation on nested relations is called* single scan processible *(linear), iff it can be computed by the abstract algebra machine with an amount of memory independent from the state of the database.*

As an example, we give a "program" which run on the algebra machine computes the nested selection mentioned above. On $Dept(dno, dname, dloc, Empl(eno, ename, sal))$ the query, written in our algebra is:

$$\sigma[\ \sigma[ename = \text{'Smith'}](Empl) \neq \emptyset\ ](Dept)$$

(notice that *all* employees are obtained from qualifying departments). A possible computation of the result using the algebra machine is shown in figure reffig-mach(b). Here we used two local variables (i.e. additional memory to store them): 'found' is a boolean flag set when the nested selection hits, and 'temp' is used to store a position of the output tape, viz. the one before the current output tuple. If a department fails the selection, we remove it by simply rewinding the tape and overwriting with either the next department or an end-of-tape mark.

3 Results

Now we look at the individual operations on nested relations and combinations of them and state whether they are single pass processible, under what conditions they are, or not.

Theorem 1 *The dyadic operations of the algebra (union, difference, product) and the nest operation are not single scan processible. So are the derived ones like intersection, join, division, etc.*

We do not give the detailed proofs here, they can be found in [Sch88]. Union might be considered a linear operation if we omit duplicate elimination as with projections. Otherwise it requires sorting like difference, intersection and nesting do, hence they are not linear. As input relations for a product can only be read one after the other from the input tape, we would need storage to hold the first relation when scanning the second, thus product is not linear.

Now, the interesting cases are (nested) selections and projections, with their variety of nested subexpressions, and unnesting. Let us begin with projections. In our algebra, projection is used to eliminate some columns of tables, but moreover, we use projection as one of the entry points for nested expressions. For instance, we can project (from the department relation) the employee subrelation as it is stored, but also the result of an arbitrary algebraic expression applied to it. This way we can obtain a new subrelation holding e.g. only the employees named Smith, or the join of the employee subrelation with a children relation, and the like[5]. Therefore, not all projections can be linear. This is the reason why we use the term single pass rather than the classical single table operation. One possible restriction is that only *linear* expressions may be nested inside a projection, but we need others. Consider the fact that more than one expression may be applied to a subrelation to construct several new subrelations. If all of these are linear, they can be computed in parallel while scanning the input, however, as subrelations are written one after the other on the output tape (as on the input), we would have to collect the results of all but the first nested subexpression until the first is written onto the output tape completely. Hence, what can be computed in a single scan is

Theorem 2 *A (nested) projection is linear, iff at most one subexpression is applied to each subrelation and all of these are linear.*

As subrelations come one after the other on the input (and output) tape, we can compute the nested subexpressions one after the other. If they are all linear, we need $\max_i(N_i)$ units of additional memory, where N_i is the requirement of nested subexpression i.

Obviously selections with standard "1NF-style" conditions are linear. However, in the nested case we have set comparisons and nested subexpressions. This means we can select tuples based on the (set) comparison between the results of two nested subexpressions, e.g. departments where the set of all employees named Smith equals (or not equals ...) the empty set, or (equals or not equals or) contains ... the set of employees making more than 50k.[6] Selection in our algebra is a second entry point for nesting of expressions. A first observation is that comparing two arbitrary sets on equality or containment is not a single scan operation (it introduces join complexity). So the only possible set comparisons in linear selections are those between a set (expression or subrelation) and a (set) *constant*, i.e. a set whose value is explicitely given in the query formulation, e.g. the empty set $\emptyset$. Then the space needed to hold the value of one of the two sets is given in the query tree!

Theorem 3 *A (nested) selection is linear, iff it is built according to the following rules:*

- *arbitrary comparisons on atomic attributes,*

[5] $\pi[\ldots, New := \sigma[ename = \ldots](Empl)](Dept)$ or $\pi[\ldots, New := Empl \bowtie Children](Dept)$

[6] we already had the first query, the second one looks like $\sigma[\ \sigma[ename = \ldots](Empl) \supseteq \sigma[sal > \ldots](Empl)\ ](Dept)$

- *element tests ($\in$) on subrelations (or nested subexpressions)*
- *set comparisons between constants and subrelations (or nested subexpressions),*
- *logical conectives of these ($\wedge, \vee, \neg$),*

where all nested subexpressions are linear.

Interestingly, we find no restriction on the number of nested subexpressions per subrelation as we had for projections. This is due to the fact that we can compute several linear operations in parallel as long as we do not have to store the results of them (cf. projections above). For selections it is sufficient to keep a boolean flag indicating the current truth-value of every subcondition. We need to remember only one position of the output tape (for the outermost nested selection).

Before we look at the unnest operation, we analyse the use of so-called *dynamic constants* [SS86], i.e. values of higher level attributes inside a deeply nested subexpression. As an example, consider our department relation and assume that we have the employee number of the department's manager (*mno*) as an atomic value. Then $\pi[dname, \pi[ename](\sigma[eno = mno](Empl)](Dept)$ retrieves a list of department names together with (as a subrelation) the names of their managers. The value of *mno* is constant w.r.t. the subrelation *Empl*, thus we could use it in the nested selection condition. We can permit dynamic constants in linear operations provided that the values of these constants do not require storage space depending on the extension. As this is only the case with set (i.e. relation) valued attributes, we have

Theorem 4 *The use of atomic dynamic constants is permitted in linear selections and projections.*

For a given query representation we can determine in advance which attribute values are needed in deeper levels of the nesting hierarchy and save their values in some intermediate storage. The size of that storage depends only on the query and the schema (i.e. the maximal attribute length).

Dynamic constants are also the key to the unnest operation: unnesting means reducing the hierarchical structure of the input relation by repeating attribute values with all subtuples of the unnested subrelation.

Theorem 5 *Unnesting a subrelation S of a relation R is a linear operation, iff S is the only subrelation of R (on that level).*

The values of all atomic attributes of R can be held as dynamic constants for the scan of S and output together with the attributes of S repeatedly. We need additional memory to store one "atomic fragment" of R-tuples. A second subrelation would require extension-dependent storage space.

While selections and projections need extra memory for tape positions and boolean flags for parts of selection conditions, i.e. in an amount primarily determined by the query, dynamic constants and unnesting require space depending on the schema (the domains of the atomic attributes) of the involved relations. Thus, unnesting and dynamic constants are linear operations w.r.t. a slightly more sophisticated notion of linearity.

Finally, we are interested in combinations (compositions) of linear operations. If we use a query language where the model is closed under its operations, we can construct complex expressions by applying more operations to the result of previous operations. In such a sequence of operations, if all operations are linear, is the combination linear too? We know from classical query optimization that given two subsequent operations we can either (i) compute the first one and obtain an intermediate result on which we apply the second operation subsequently, or (ii) compute both of them in a "pipelined" mode, apply the first operation to the first input tuple and feed the result directly to the second operation (see e.g. [JK84]). The second alternative is more efficient in general since we avoid storage of potentially large intermediate results and multiple scans. Therefore, we should be able to combine several linear operations into one linear expression and compute it in pipelining mode.

Theorem 6 *Any composition of linear operations yields a linear operation.*

While we can combine linear operations in an arbitrary way due to the above result, we are interested in a "canonical linear expression" in order to develop guidelines for the implementation of a low-level (i.e. linear) nested relational query processor and for an optimizer. Using algebraic equivalence rules on our nested algebra we can apply transformations similar to the cascading of selections and projections, commutation of projections and selections, and so on [Sch86]. If we only consider (nested) selections and projections, such a canonical form is:

$$\sigma[F_2](\pi[L](\sigma[F_1](R))),$$

where F_1 and L may contain nested linear subexpessions of this type in turn, F_2 is a selection condition which refers to the results of expressions in L (otherwise it could have been commuted with the projection). F_1 and L may have several nested subexpressions on the same subrelation, however, only one of them may occur in L. Typical examples of conditions found in F_2 are of the form "result$\neq \emptyset$", i.e. tuples that do not have any matches in a nested subexpression of L are discarded, like in

("retrieve a list of department (names) each with a list of employees (names) making more than 50k, drop departments without any such employees"). In addition to the above canonical form we can have unnest operations at any place, however, often it will be sufficient to unnest as a last step:

$$(\mu[\ldots])^*\sigma[F_2](\pi[L](\sigma[F_1](R))),$$

where $(\mu[\ldots])^*$ means a sequence of unnests (each of which applies to the only subrelation).

Conclusion

We identified the class of single scan processible operations in a nested relational query language. We used the nested algebra of [SS86] in the discussion, however, the results apply to all query languages with similar query facilities. The class of linear operations plays an important role if we

- implement a nested relational DBMS
  where we should try to realize the full functionality of that class within the lowest level of query processing algorithms, the canonical linear expression may be used in defining the interface of such a component,
- design an optimizer
  which should map as many queries as possible to linear expressions, as this helps in reducing processing costs by avoiding duplicate scans.

Concerning the DASDBS prototype, we find a subclass of linear expressions very close to the canonical form at the low-level query processing layer (CRM, see [DPS86, PSS+87]). Algebraic optimization in our nested relational algebra has been studied in [Sch86, SPS87, Sch88], where we found that it is exactly this class of queries that is needed to efficiently support what would be select-project-join queries on an equivalent flat relational database.

References

[ABC+76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffith, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[CAB+81] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Shultz, I. L. Traiger, B. W. Wade, and R. A. Yost. History and evaluation of System R. *Communications of the ACM*, 24(10):632–646, October 1981.

[DGW85] U. Deppisch, J. Günauer, and G. Walch. Storage structures and addressing techniques for the complex objects of the NF^2 relational model. In A. Blaser and P. Pistor, editors, *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications*, pages 441–459, Karlsruhe, 1985. IFB 94, Springer. (in German).

[DPS86] U. Deppisch, H.-B. Paul, and H.-J. Schek. A storage system for complex objects. In *Proc. Int. Workshop on Object-Oriented Database Systems*, pages 183–195, Pacific Grove, September 1986.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–154, June 1984.

[Loh88] G.M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, June 1988. ACM, New York.

[PSS+87] H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, and U. Deppisch. Architecture and implementation of the Darmstadt database kernel system. In *Proc. ACM SIGMOD Conf. on Management of Data*, San Francisco, 1987. ACM, New York.

[Sch86] M. H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *ICDT '86: Int. Conf. on Database Theory, Rome*, pages 380–396. LNCS 243, Springer, Berlin, Heidelberg, 1986.

[Sch88] M. H. Scholl. *The Nested Relational Model —Efficient Support for a Relational Database Interface—*. PhD thesis, Dept. of Computer Science, Technical University of Darmstadt, 1988. (in German).

[SPS87] M. H. Scholl, H.-B. Paul, and H.-J. Schek. Supporting flat relations by a nested relational kernel. In *Proc. Int. Conf. on Very Large Databases*, pages 137–146, Brighton, September 1987. Morgan Kaufmann, Los Altos, CA.

[SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, June 1986.

[SWKH76] M. R. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

Query Processing in a Nested Relational Database System

José A. Blakeley Anand Deshpande

Computer Science Department
Indiana University
Bloomington, IN 47405, USA
{blakeley,deshpand}@iuvax.cs.indiana.edu

Abstract

This paper presents a query processing architecture for a nested relational database system. The focus of the paper is a brief description of its novel query processing features: (a) extensive use of structured tuple-ids that allow computation in the tuple-id-world rather than in the value-world, (b) application of the rule-based query optimization approach to provide the extensibility required in the system, (c) main-memory query optimization similar to compiler-like optimizations, and (d) efficient computation of joins.

1 Query processing

We are addressing the problem of query processing for applications involving complex objects. We started by studying the problem in the context of the nested relational model. However, our long-term objective is to build a system that can be extended to support facilities of object-oriented databases. In our quest to implement an efficient nested relations/complex-object database system we found that query optimization was a non-trivial task because:

- The nested relational algebra operators involving select, nest and unnest lack the nice commutative and associative properties of the relational algebra, making the optimization process along the lines of algebraic relational optimization difficult [12].
- Most nested algebras define projections and joins at the top-most level in the structure. Mathematically it is possible to transform deeply-nested attributes to the top-most level by successive restructuring operations (i.e., nest and unnest). Such restructuring operators are expensive and should be avoided whenever possible.
- Object-oriented databases use object-ids to identify objects. Unfortunately, object-ids are generated in an ad-hoc fashion. Static ad-hoc object-ids make them unsuitable for optimization.
- Queries in object-oriented databases involve extensive navigation making such queries difficult to optimize.

Clearly, query processing for nested relations and complex objects is not a straightforward extension of relational query optimization. We are currently developing **ANDA** (Architecture for Nested Database

Applications), a prototype system that supports the nested relational model. The salient features for query evaluation in **ANDA** are described as follows:

- Structured tuple-ids: A judicious design of tuple-ids and their exploitation for query processing. Such tuple-ids facilitate the design of new data structures (i.e., **VALTREE** and **RECLIST**) that support efficient retrieval of value-driven queries in the nested relational model.

- Tuple-id based query processing: Query processing in **ANDA** follows the basic theme that the user specifies queries in terms of values – the so called "value-world." As it is more efficient to process queries on tuple-ids in main-memory, the query is converted to the "tuple-id" world. After processing tuple-ids in the tuple-id world, the tuple-ids are materialized to return to the value-world. A typical query in **ANDA** involves a sequence of "value → tuple-id → value" (VTV) cycles.

- Main-memory query processing: An access language that permits query processing on tuple-ids in main memory unlike traditional query processing which uses temporary relations on disk. This opens up new opportunities for query optimization not widely explored in previous database research on query optimization for the relational model. In particular, by doing extensive query processing in main memory on tuple-ids, it is possible to apply techniques commonly used in optimization of code in compilers (e.g., data flow analysis).

- Rule-based optimization: We build on the ideas of rule-based query optimization for extensible database management systems [5,6] to provide the desired extensibility in our system. We have already mentioned that the commutative and distributive properties of the nested relational algebra do not hold as generally as in the relational algebra. Instead of designing yet another nested algebra we chose to incorporate basic algebraic heuristics into rules that are used to generate query plans. In our system, this is included in what we call the "algebraic rule-base."

  We want to support in the future, object-oriented query processing features. Hence, it is likely that extensions, or even a new access language might be required. Therefore, rather than relying on a single low-level interface for nested relations as it is the case in relational systems (e.g., System R's RSS [2]) we support several possible implementations of the access language. We provide this flexibility by encoding the implementation of the access languages in what we call the "access language rule-base."

- Efficient joins: Single-value join operations on deeply-nested attributes are performed very efficiently using the **VALTREE**. In particular, the **VALTREE** can be seen as a generalization of a join index [13].

So far, in **ANDA** we have addressed only set-oriented query processing issues. The nested relational model is "purely" value-based. By extending the model to allow tuple-id values within nested relations the model is no longer value-based. In such an extended model, it is important to support navigation through tuple-ids. We plan to address navigational query processing in the future.

2 The query processing environment

Figure 1.a illustrates the query processing stages in our system. During parsing, a Nested SQL[1] statement

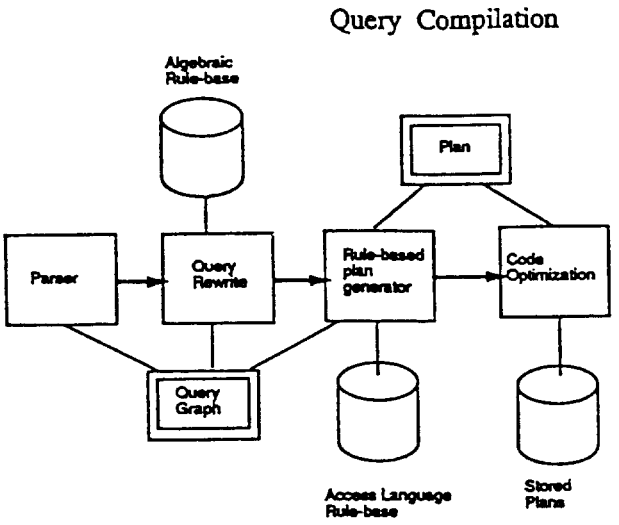


Figure 1.a: The query processing architecture

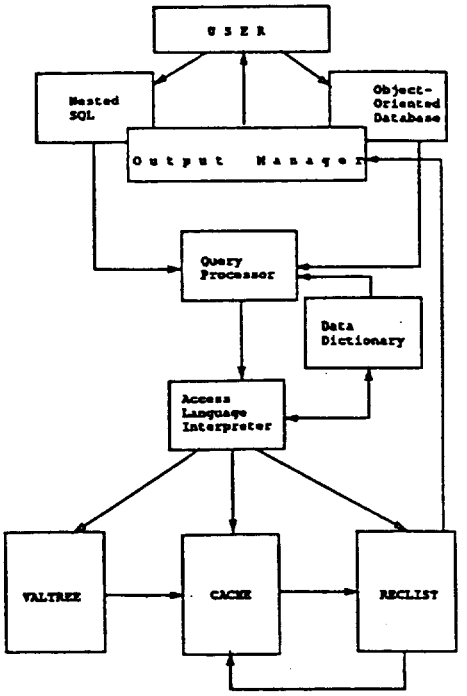


Figure 1.b: The ANDA architecture

is translated to an internal graph representation. The graph representation is an annotated graph resulting from the parse tree similar to the query graph model in Starburst [8,9]. The query rewrite component takes the query graph resulting from parsing and transforms it into a graph leading to a better plan using the algebraic rules. The rule-base plan generator takes the query graph that results from the query rewrite component and, using the access language rule-base, generates a program in the ANDA access language that executes the query. Data-flow analysis similar to the one performed by optimizing compilers [1] is performed on the plan produced by the plan generator. The resulting plan is then stored for future execution.

ANDA's run-time system, illustrated in Figure 1.b, consists of three basic components – the VALTREE, the CACHE and the RECLIST. The VALTREE, stores an efficient mapping from values of the database to a set of tuple-ids that correspond to all occurrences of the values in the entire database. The RECLIST provides a mapping from tuple-ids to values. Retrievals on the VALTREE convert the query from the value-world to the tuple-id world. The RECLIST is used to convert from the tuple-id world to the value world. The CACHE which is the main-memory component of the database is used to manipulate tuple-ids in the tuple-id world.

The structure of tuple-ids in ANDA uses relation names tagged with subscripts and superscripts [3,4]. The subscripts go "down" tuples and superscripts go "across" the components (i.e., attributes and nested structures). The organization of tuple-ids for hierarchies of nested relations has the following properties: (a) every value has a unique tuple-id, (b) given a deeply-nested tuple-id, it is possible to determine the tuple-ids of other components of the subtuple and the tuple-ids of the super-tuple and (c) given two

[1]We are currently developing a Nested SQL query language interface for nested relations similar to SQL/NF [10,11] and Laurel [7] as part of this project.

tuple-ids it is possible to compare the tuple-ids to determine if they belong to the same tuple or sub-tuple.

3 An example

We now show an example that will serve to illustrate a typical compilation of a query. The example is based on the following nested relational database scheme describing courses and students.

```
Course = (cname, dept, cno, credits, Prerequisite, Section)
  Prerequisite = (cno)
  Section = (secid, term, instructor, Enrollment)
    Enrollment = (sno, grade)
Student = (sname, sid, class, school)
```

Suppose we want to find all the courses taken at the same time by Jones and Smith. This query can be posed in Nested SQL as follows:

```
SELECT cno, cname, Section.term FROM Courses C
WHERE C.Section.Enrollment.sno
CONTAINS (SELECT sno FROM Student
          WHERE sname = "Jones" OR sname = "Smith")
```

This Nested SQL query will be compiled into the following ANDA program:

```
value-world:
  1.  vt-retrieve(TEMP1,Name,Jones,sname,Student,relation)
  2.  vt-retrieve(TEMP1,Name,Smith,sname,Student,relation)
tuple-id-world:
  3.  cache-union(TEMP1,S**)
  4.  cache-transform(TEMP1,S*a)
value-world:
  5.  tuple-id-to-value(TEMP1,TEMP2)
  6.  cache-explode(TEMP2)
  7.  vt-retrieve(TEMP3,Sid,cache-pop(TEMP2),sno,Course,relation)
  8.  vt-retrieve(TEMP3,Sid,cache-pop(TEMP2),sno,Course,relation)
tuple-id-world:
```

For a current description of each of the CACHE functions mentioned in the above program, the reader is referred to Deshpande [3]. The names TEMP1, TEMP2, TEMP3, and RESULT refer to named stacks. This is similar to the temporary storage locations generated in the code generation stage of a compiler. This program shows several key ideas behind query compilation in our system.

- As we mentioned before, the simplest Nested SQL query gets compiled into a program containing at least one VTV cycle. In this case our example contains two such cycles.
- Statement 1, 2, 5, 7, 8, and 18 involve functions that retrieve values from secondary storage (i.e., VALTREE and RECLIST). Such statements should be minimized when possible.
- Blocks of statements 3-4 and 9-17 represent cache operations performed in main memory. Two observations can be made at this point: (1) it is possible to eliminate some of the cache operations. For instance, by performing the transformation in statement 4 after the vt_retrieve it is possible to avoid the cache union and explode operations of statements 3 and 6; and (2) it is possible to change the semantics of the query to obtain all courses taken by Smith and Jones not necessarily at the same time by using statement 9' rather than 9 which changes the "window" of the intersection operation.

4 Current status of the system

A prototype of ANDA that implements the low-level access language has been implemented in the C programming language on a Sun 3/60 running Unix. All routines that operate on the VALTREE and the RECLIST perform actual secondary storage accesses. A Nested SQL interface has been implemented and so far a direct translation from the parse tree to the access language is performed using the access language rule-base. The implementation of the algebraic rule-base is currently under development. A rudimentary data-flow analysis is performed on the access language programs currently generated. Additional "compiler-like" optimizations are also under development.

In this paper we have briefly outlined ANDA's query processing system whose main features are: (a) extensive use of structured tuple-ids that allow computation in the tuple-id-world rather than in the value-world, (b) application of the rule-based query optimization approach to provide the extensibility required in the system, (c) main-memory query optimization similar to compiler-like optimizations, and (d) efficient computation of joins.

References

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers, principles, techniques, and tools.* Addison-Wesley Publishing Company, Reading, Ma, 1986.

[2] ASTRAHAN, M., BLASGEN, M., CHAMBERLIN, D., ESWARAN, K., GRAY, J., GRIFFITHS, P., KING, W., LORIE, R., MCJONES, P., MEHL, J., PUTZOLU, G., TRAIGER, I., WADE, B., AND WATSON, V. System R: Relational approach to database management. *ACM Transactions on Database Systems 1*, 2 (June 1976), pp. 97–137.

[3] DESHPANDE, A. *An Implementation for Nested Relational Databases.* PhD thesis, Indiana University, Bloomington, Indiana 47405, Expected June 1989.

RANDOMIZED ALGORITHMS FOR OPTIMIZING LARGE JOIN QUERIES

(Extended Abstract)

Yannis E. Ioannidis
Younkyung Kang
Department of Computer Sciences
University of Wisconsin, Madison
({yannis,youn}@cs.wisc.edu)

1. INTRODUCTION

The key to the success of a Database Management System (DBMS), especially of one based on the relational model [Codd70], is the effectiveness of the query optimization module of the system. The input to this module is some internal representation of an ad-hoc query q given to the DBMS by the user. Its purpose is to select the most efficient algorithm to access the relevant data and answer the query. Specifically, for the relational model, a *strategy* to answer q is a sequence of relational algebra operators applied to the relations in the database that eventually produces the answer to q. Let S be the set of all such strategies. Each member s of S has an associated cost $c(s)$ (measured in terms of CPU and I/O time). The goal of any optimization algorithm is to find the member s_0 of S that satisfies

$$c(s_0) = \min_{s \in S} c(s).$$

Whenever the cardinality of S is large, performing an exhaustive search of S is impossible. Which part of S is worth exploring is the implementor's decision. The smaller the explored space is, the higher the probability that the optimum is missed, and the faster the optimization algorithm runs.

Query optimization has been studied quite extensively theoretically [Aho79, Rose80], in the context of centralized DBMSs [Wong76, Seli79, Mack86a], and in the context of distributed systems [Epst78, Bern81, Mack86b]. Good surveys exist also [Jark84, Kim86].

The unit of optimization in most existing DBMSs is a single query. Each query involves a small number of relations (e.g., less than 10). Hence, even though the number of alternative access plans to answer a query grows exponentially with the number of the relations in the query, this number is relatively small. Most existing query optimizers perform an exhaustive search over the space of alternative access plans, and whenever possible, use heuristics to reduce the size of that space.

The unit of optimization in most existing DBMSs is a single query. Each query involves a small number of relations (e.g., less than 10). Hence, although the number of alternative strategies to answer a query grows exponentially with the number of the relations in the query, this number is relatively small. Most existing query optimizers perform an exhaustive search over the space of alternative strategies, and whenever possible, use heuristics to reduce the size of that space.

The above picture is expected to change, however, in future DBMSs. The most important of these changes are identified below.

participating queries that can speed up execution if they are taken into account. Finally, the number of nonrecursive queries that are equivalent to a recursive one is arbitrarily large. These nonrecursive queries share several common subexpressions, since each is equivalent to repeatedly applying the same query several times. Hence, the size of the resulting strategy space can be arbitrarily large as well. One of the solutions proposed to face the unprecedented size of strategy spaces is to use randomized algorithms, which are discussed in the following subsection.

2. RANDOMIZED ALGORITHMS FOR QUERY OPTIMIZATION

Two randomized algorithms have been recently proposed for query optimization on large strategy spaces: one based on the *Simulated Annealing* algorithm [Ioan87], and another based on *Iterative Improvement* [Swam88].

2.1. Simulated Annealing

Simulated annealing is a *Monte Carlo* optimization technique proposed by Kirkpatrick, Gelatt, and Vecchi for complex problems that involve many degrees of freedom [Kirk83]. Such problems are modeled by a state space, where each state corresponds to a solution to the problem. A cost is associated with each state, and the goal is to find the state associated with the globally minimum cost. For complex problems with very large state space, exhaustive exploration of all the states is impractical. Probabilistic hill climbing algorithms, such as simulated annealing, attempt to find the global minimum by traversing only part of the state space. They move from state to state allowing both downhill and uphill moves, i.e., moves that reduce and moves that increase the cost of the state respectively. The purpose of the latter kind of moves is to allow the algorithm to escape from local minima it may occasionally encounter. In simulated annealing, the uphill moves are controlled by a parameter T, the temperature. The higher T is, the higher the probability an uphill move is taken. As time passes, T decreases, and at the end, when the system is "frozen" ($T = 0$), the probability of making an uphill move is negligible. Many theoretical investigations have been performed on the behavior of the simulated annealing algorithm [Rome84, Haje85]. It has been shown that, under certain conditions satisfied by various parameters of the algorithm, as T approaches 0, the algorithm converges to the state of global minimum cost. Details about this algorithm can be found elsewhere [Kirk83, Ioan87].

2.2. Iterative Improvement

Iterative improvement is another Monte Carlo optimization technique, applied on similar problems (i.e., similar state space structure and cost functions) as simulated annealing. It is not a hill climbing algorithm, because no uphill moves are taken. All the moves are downhill. Given an arbitrary state, the algorithm moves to a local minimum that is close to that state by a series of downhill moves. This is done for several randomly chosen states, so that the algorithm is not stuck in any local minimum. The whole process stops after it has been repeated enough time, which can be measured in several ways (e.g., running time, number of local minima reached, number of states visited). A simple stochastic analysis shows that as time approaches ∞, the algorithm converges to the state of global minimum cost. Details about this algorithm can be found elsewhere [Swam88].

2.3. Comments

The successful application of simulated annealing and iterative improvement on several optimization problems, together with their theoretical foundation and their elegant simplicity, has been the primary motivation to devise query optimization algorithms based on them. Although the structure of the algorithms is problem-independent, some of their parameters depend on the particular problem in hand. Probably the most important of these is the state space S. For query optimization, S is the set of strategies one can apply to answer a given query q. The state space can be enhanced according to any special characteristics of q, or it can be made smaller heuristically by removing strategies that are likely to be suboptimal. For simulated annealing, elimination of states must be done with great caution, because the reduced state space has to remain strongly connected. Otherwise, the optimal state may not be reachable from the initial state.

Both simulated annealing and iterative improvement are especially well suited to optimization problems with large search spaces and with cost functions that manifest a large number of local minima. If the number of local minima is small, which is the case in conventional query optimization, these algorithms are inappropriate. This is not the case, however, when DBMSs are used in new application domains, which require optimization of queries involving many relations, global optimization, or recursive query optimization.

3. GETTING THE BEST OF BOTH WORLDS

3.1. Performance Evaluation

We have performed a comprehensive study of the performance of Simulated Annealing and Iterative Improvement for a variety of select-project-join queries. Our study is similar in nature to that of Swami and Gupta [Swam88]. There are three distinct differences though. First, their state space consists of "left-deep trees" only, whereas we include "bushy trees" as well. Second, they examine only one join algorithm, namely hash-join, whereas we have experimented with two of them, namely nested-loops and merge-scan. Third, they incorporate one transition rule from a state to its neighbors, namely exchanging the position of two relations in the query tree, whereas our transition rules are based on algebraic properties of joins as well as switching join algorithms for a join. All these differences make the state space that we have to deal with larger. They also make certain transitions from one state to the other more expensive, e.g., sort-order of relations must be propagated up the query tree for possible use in a higher level merge-scan join.

Both algorithms have been implemented in C and tested under Unix on a Sun-4. Our results qualitatively confirm those of the previous study [Swam88], but they also extend them in several insightful ways. Iterative improvement very quickly reaches a good local minimum, beyond which the improvement in the cost of the solution is not dramatic. On the other hand, simulated annealing spends much time in states with very high cost. Nevertheless, it eventually finds its way into low cost states and converges to one that is always better, i.e., cheaper, than the one that iterative improvement has found. Figure 1 shows the cheapest state visited by the two algorithms as a function of time, for the average run. The y-axis represents the ratio of the state cost over the best cost found among all runs of both algorithms. The specific diagrams are for 40 join tree-queries, but similar results have been observed on other types of queries for sizes between 10 and 100 joins.

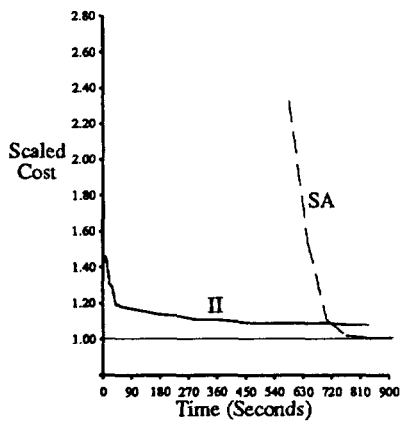


Figure 1: Cheapest strategy found as a function of time.

Contrary to the conclusions of the work of Swami and Gupta [Swam88], the above figure suggests that simulated annealing has the ability to outperform iterative improvement if it is run long enough. One must admit that the observed difference has been disproportionally small with respect to the extra time spent by simulated annealing. Observed differences in final strategy cost have been between 10% and a factor of 2, whereas those in optimization time have been between a factor of 5 and a factor of 10. Nevertheless, simulated annealing is still valuable on the following basis. Both algorithms are only useful for very large queries. The execution time of such queries is expected to be very long, so it is unlikely that they will be submitted interactively. Most likely, these queries will be compiled (thus, optimized only once), and executed multiple times. Hence, even small improvements in the cost of the strategy of choice can result in big savings in the amortized cost of all runs of the query. The additional time that simulated annealing needs to achieve the improvements in the quality of the result may well be justified.

3.2. The Shape of the Cost Function

The consistency with which diagrams like that of Figure 1 were observed for all types of queries and sizes that we tried indicated to us that the algorithms' behavior must be a result of some fundamental features of the state space we were exploring and its cost distribution. We decided to investigate the shape of the cost function in the state space. The size of the latter is prohibitive of any attempt of an exhaustive search of it. Hence, randomization was employed again to overcome this problem. The following experiments were performed and the

following results were observed:

- We randomly generated states in the state space and calculated their costs. We only performed experiments of 10 and 40 join queries, and we generated 50-100 million states in each case. In the 10 join queries, the minimum that was found was the cost of the state that simulated annealing had converged to. In the 40 join queries, the minimum that was found was about 4 times the cost of the state that simulated annealing had converged to. The conclusion from this experiment is that the area of good, i.e., cheap, states is very small compared to the total state space, and that simulated annealing finds a plan that is very good, i.e., the likelihood of the existence of a state with cost lower than the one simulated annealing found is very slim.
- We performed random walks in areas of low cost states. Our purpose was to get a feeling for the number of good local minima that exist and their mutual distance. The random walks consisted of alternating series of uphill moves and series of downhill moves. Each series of downhill moves ended in a local minimum. Each series of uphill moves ended when the state cost exceeded a prespecified limit (so that it is ensured that we remain in areas of low cost states). On the average, 70% of all visited local minima had distinct costs. Also, these distinct local minima were close to each other; for example, for 40 join queries and not much variance in the sizes of the relations, the average distance between local minima was 25 states (in a space of more than 2^{40} states). This implies that there is a relatively small area of low cost states, which contains a large number of local minima, which are very close to each other. Interestingly, we performed the same experiment in search for local maxima. Surprisingly, for a rather extensive area of low to medium cost states, we found no local maxima. The only local maxima we were able to identify were states with very high cost.

The above results lead to the following conclusion regarding the shape of the cost function on the state space.

The shape of the cost function resembles a cup, with some relatively small variations at the bottom.

In other words there is a small area of states with low costs, surrounded by the remaining states with increasingly higher costs. There is relatively small variation among the costs in the low cost area, but enough to make exploration of that area worth while. Pictorially, and for a one-dimensional function, the situation is shown in Figure 2.

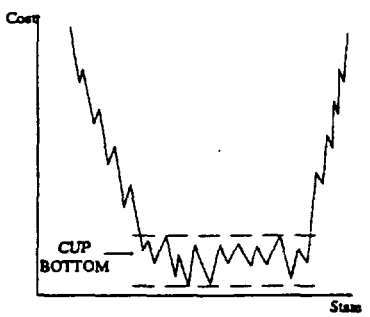


Figure 2: Shape of cost function.

The shape of Figure 2 is further validated by the results of the observed behavior of the algorithms shown in Figure 1. Iterative improvement starts from a randomly generated state and moves down to a local minimum repeatedly. Because of the cup shape, very soon it reaches at a local minimum that is at the bottom of the cup (this is also helped by the fact that there are no local maxima in low cost areas). However, because of the small size of the cup bottom, it is unlicely that iterative improvement will randomly pick a state in the cup bottom; so, not many local minima in the low cost area are explored (this has also been verified by measurements of the time that iterative improvement spends in low cost states). On the other hand, simulated annealing spends much time in high cost areas, but when the temperature starts cooling down it eventually is forced to the cup bottom, which it explores quite extensively until freezing. This explains why it takes time for simulated annealing to visit a low cost state. It also explains why, by spending more time around the cup bottom (with low temperature), simulated annealing eventually converges to a state that is superior to the one that iterative improvement converges to, i.e., to one of lower cost.

The above results are not in agreement with those of Swami and Gupta [Swam88], who observed that Simulated Annealing was never superior to Iterative Improvement, independent of the amount of time that was given to it. This difference, however, is very easily explained by the difference in the state space and in the transformations that were used in the two studies, primarily the latter. Swami and Gumpta used only one transformation, namely exchanging the position of two relations in the access plan. This generates neighbors that have large differences in their cost, which makes the shape of the cost function much less smooth (not a cup), and therefore Simulated Annealing does not have the opportunity to spend much time in a low cost area. In our experiments, by using different connections among the states, we managed to improve the algorithm's performance and produce superior strategies.

3.3. Hybrid algorithm

The above observations on the shape of the cost function and the behavior of the two algorithms inspired a new *Hybrid* algorithm to take advantage of the specific properties of query optimization. Hybrid is a combination of Iterative Improvement and Simulated Annealing. First it performs Iterative Improvement for a small period of time. This gives the opportunity to the algorithm to start at a random state and move down at the cup bottom for a few times. Then the algorithm performs Simulated Annealing. The initial state was the best state visited in the iterative improvement phase. The initial temperature is low enough so that the algorithm cannot escape from the cup bottom, but is high enough so that adequate time is given to it to extensively explore the cup bottom.

Running the Hybrid algorithm on the same queries that Iterative Improvement and Simulated Annealing were run yielded the results we expected. Hybrid almost always converged to a better plan than either Iterative Improvement or Simulated Annealing did, and for the times it did not, it was very close to the better of the two (usually, the one of Simulated Annealing). It also consumed much less time than Simulated Annealing (by about a factor of 3 on the average), primarily because it did not waste time in the useless high cost areas. A representative graph of the behavior of all three algorithms, in terms of the best plan found during the course of the average run is shown in Figure 3. The specific graph shows again scaled cost for a 40 join tree-query.

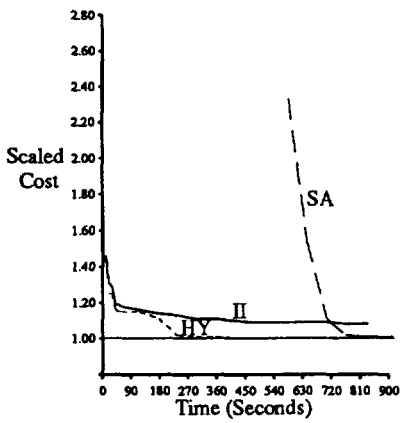


Figure 3: Cheapest strategy found as a function of time.

4. CONCLUSIONS

The above results indicate that one can take advantage of the specific properties of the problem of query optimization and the cost functions it gives rise to. Algorithms like Hybrid can be used, which converge in a relatively short amount of time to close-to-optimal strategies even for large queries.

We are currently performing more experiments with the three algorithms on several additional queries to validate our observations, and to confirm the extent of their applicability. We are also extending the algorithms so that they can deal with queries that involve unions also. In that case, the state space is even larger, but some very preliminary results indicate that one can take advantage of similar characteristics as for join-only queries and develop fast randomized query optimization algorithms.

5. REFERENCES

[Aho79]
Aho, A., Y. Sagiv, and J. Ullman, "Equivalences Among Relational Expressions", *SIAM Computing Journal* **8**, 2 (May 1979), pp. 218-246.

[Banc86]
Bancilhon, F. and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", in *Proc. of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pp. 16-52.

[Bern81]
Bernstein, P. A., N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)", *ACM TODS* **6**, 4 (December 1981), pp. 602-625.

[Codd70]
Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", *CACM* **13**, 6 (1970), pp. 377-387.

[Epst78]
Epstein, R., M. Stonebraker, and E. Wong, "Distributed Query Processing in a Relational Data Base System", in *Proc. of the 1978 ACM-SIGMOD Conference on the Management of Data*, Austin, TX, May 1978, pp. 169-180.

[Gran81]
Grant, J. and J. Minker, "Optimization in Deductive and Conventional Relational Database Systems", in *Advances in Data Base Theory* **Vol. 1**, edited by H. Gallaire, J. Minker and J. M. Nicolas, Plenum Press, New York, N.Y., 1981, pp. 195-234.

[Haje85]
Hajek, B., *"Cooling Schedules for Optimal Annealing"*, unpublished manuscript, January 1985.

[Ioan87]
Ioannidis, Y. E. and E. Wong, "Query Optimization by Simulated Annealing", in *Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data*, San Francisco, CA, May 1987, pp. 9-22.

[Jark84]
Jarke, M., J. Koch, and ""Query Optimization in Database Systems", , *ACM Computing Surveys* **16**, 2 (June 1984), pp. 111-152.

[Kim86]
Kim, W., D. Reiner, and D. Batory, *Query Processing in Database Systems*, Springer Verlag, New York, NY, 1986.

[Kirk83]
Kirkpatrick, S., C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing", *Science* **220**, 4598 (May 1983), pp. 671-680.

[Kris86]
Krishnamurthi, R. and C. Zaniolo, "Safety and Optimization of Horn Clause Queries", in *Preprints of the Workshop on Foundations of Deductive Databases and Logic Programming*, Washington, DC, August 1986, pp. 518-543.

[Mack86a]
Mackert, L. F. and G. M. Lohman, "R* Validation and Performance Evaluation for Local Queries", in *Proc. of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pp. 84-95.

[Mack86b]
Mackert, L. F. and G. M. Lohman, "R* Validation and Performance Evaluation for Distributed Queries",

in *Proc. 12th International VLDB Conference*, Kyoto, Japan, August 1986, pp. 149-159.

[Rome84]
Romeo, F., A. Sangiovanni-Vincentelli, and C. Sechen, "Research on Simulated Annealing at Berkeley", in *Proc. 1984 IEEE International Conference on Computer Design*, Port Chester, N.Y., October 1984, pp. 652-657.

[Rose80]
Rosenkrantz, D. J. and H. B. Hunt III, "Processing Conjunctive Predicates and Queries", in *Proc. 6th International VLDB Conference*, Montreal, Canada, October 1980, pp. 64-72.

[Seli79]
Selinger, P. et al., "Access Path Selection in a Relational Data Base System", in *Proc. of the 1979 ACM-SIGMOD Conference on the Management of Data*, Boston, MA, June 1979, pp. 23-34.

[Sell86]
Sellis, T. K., "Global Query Optimization", in *Proc. of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pp. 191-205.

[Swam88]
Swami, A. and A. Gupta, "Optimization of Large Join Queries", in *Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data*, Chicago, IL, June 1988, pp. 8-17.

[Wong76]
Wong, E. and K. Youssefi, "Decomposition - A Strategy for Query Processing", *ACM Transactions on Database Systems* 1, 3 (September 1976), pp. 223-241.

Research in Optimization of Large Join Queries

Arun Swami
Computer Science Department, Stanford University, Stanford, CA 94305
arun@polya.stanford.edu

Keywords: search, large join queries, statistical distributions, combinatorial optimization techniques, optimization heuristics, validation

Introduction

Much effort has been devoted to developing good plans for executing queries in relational database systems [3]. These plans are termed *query evaluation plans* (QEPs). Current query optimizers normally expect to process queries involving a small number of joins (less than 10 joins). The search spaces are small enough that the use of search techniques such as the System R dynamic programming algorithm [6] is feasible, even though the algorithm has a worst case time complexity of $O(2^N)$. However, these search techniques are inadequate for processing queries with a much larger number of joins, say, 10 to 100 joins. Such large join queries can be generated by use of multiple levels of views, applications from logic programming, and object–oriented database systems (for example, Iris [1]) and knowledge base systems that use relational databases for storage.

In our research, we are investigating the problem of optimizing Select–Project–Join queries with a large number of joins. The fundamental problem with optimizing large join queries is searching the large spaces of query evaluation plans or solutions. We note that the optimization techniques developed in this research can be adapted to processing other kinds of queries, that is, these search techniques are generally applicable.

Distributions of Query Evaluation Plans

The size of the search space grows at least exponentially as a function of the number of joins in the query. From this it does not immediately follow that the difficulty of finding a good query evaluation plan (QEP) increases as rapidly. The ease or difficulty of searching for a good QEP depends to a large extent on the proportion of good QEPs in the search space, or, in general, on the statistical distribution of the costs of QEPs. We are using the techniques of random sampling to obtain the statistical distributions of QEP costs for large join queries. Some of the results we have obtained are summarized below (see [8] for more details).

Most query optimizers use the cross product heuristic. This heuristic restricts the solution space to be searched by postponing cross products as late as possible. The intuition is that cross products are expensive and result in large intermediate results. The heuristic would be considered effective if it increased the proportion of good QEPs in the search space. We find that the number of low cost query plans sampled increases significantly when the heuristic is used. For 10 join queries, using the heuristic increases the percentage of low cost query plans from 2.4% to 79.6%; the corresponding

increase for 50 join queries is from < 0.1% to 3.8%. Thus, in absolute terms, the heuristic is not sufficiently effective for large join queries.

Even when the cross product heuristic is used, we find that the percentage of low cost QEPs decreases rapidly as the number of joins in the query increases. This holds even if the definition of 'low cost' is changed, that is, even if the acceptance threshold of query plan cost is increased. These results were verified to hold for a large number of queries with differing characteristics.

Combinatorial Optimization Techniques

The search space of QEPs for large join queries is pruned in some standard ways. Selections and projections are pushed down, the cross product heuristic is employed, and only outer linear join trees are considered. *Outer linear join trees* are binary join trees where the inner operand of each join is a base relation. However, the solution space is still too large; the problem is an NP–hard combinatorial optimization problem. Techniques such as iterative improvement and simulated annealing have been applied to combinatorial optimization problems in other areas.

Iterative improvement is essentially the greedy heuristic with multiple start states. Simulated annealing [4] has been used with great success in a number of CAD optimization problems. It has been applied to the problem of optimizing recursive queries [2]. We adapt these and other techniques to the problem of optimizing large join queries and compare them. We find that iterative improvement is clearly superior to the other combinatorial techniques, and simulated annealing is the next best algorithm. This work is described in detail in [9].

Optimization Heuristics

The power of combinatorial optimization techniques may be enhanced by combining them effectively with good heuristics. In our research, we have developed two heuristics, augmentation and local improvement. In our experiments, we also study the performance of a third heuristic proposed for large join queries, the KBZ heuristic, named after the authors Krishnamurthy, Boral, and Zaniolo. The heuristic is described in [5], and is evaluated for queries with upto 15 joins in [10].

The *augmentation* heuristic works bottom up, building a good join order a relation at a time. The heuristic tries to minimize the join selectivity in choosing the next relation in the join ordering. The idea is to keep the intermediate result sizes small over the entire join ordering. *Local improvement* works top down using the 'divide and conquer' paradigm. It decomposes the complete join ordering into many small join ordering problems, finds good solutions for the ordering subproblems, and then composes the entire join ordering out of these solutions.

We combine augmentation, local improvement, and the KBZ heuristic with iterative improvement and simulated annealing in many different ways. As an example, the augmentation heuristic can be used to provide start states for the greedy runs of iterative improvement. Our comparison of the various combinations shows that two combinations of augmentation and iterative improvement are superior to all the other optimization methods. These results are validated by using several synthetic benchmarks of queries. A more complete description of this work is in [7].

Further Research

Much work remains to be done in this area. In the research to date, we have constrained the problem in various ways. Relaxing these constraints can enhance the utility of these studies. We have restricted our attention to the hash join method; extending our comparisons to other join methods is fairly straightforward. We need to consider binary join trees other than outer linear join

trees; again, this should not be too difficult. Considering queries more complex than Select–Project–Joins is more problematic; it is no accident that most of the query optimization literature considers only this class of queries. Of course, this is partially motivated by the fact that this class of queries is a large and important class of queries.

Another interesting direction for research is to study carefully the relationships between join graphs and the good QEPs for the queries that correspond to these join graphs. Such studies may lead to a more intelligent application of the divide and conquer principle than currently done in local improvement.

Acknowledgements

This research is supported by Hewlett–Packard Laboratories under the contract titled "Research in Relational Database Management Systems" and, earlier, by DARPA contract N00039-84-C-0211 for Knowledge Based Management Systems.

References

[1] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An Object–Oriented DBMS. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[2] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 9–22, 1987.

[3] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[4] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation (Part I). Draft, June 1987.

[5] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 128–137, Kyoto, Japan, 1986. Morgan Kaufman.

[6] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1979.

[7] A. Swami. Optimization of Large Join Queries: Combining Heuristics with Combinatorial Techniques. To appear in *Proceedings of ACM-SIGMOD International Conference on Management of Data*, May 1989.

[8] A. Swami. Statistical Distributions of Costs of Evaluation Plans for Large Join Queries. Submitted to VLDB-89, January 1989.

[9] A. Swami and A. Gupta. Optimization of Large Join Queries. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 8–17, 1988.

[10] E. E. Villarreal. Evaluation of an $O(N^2)$ Method for Database Query Optimization. Master's thesis, The University of Texas at Austin, May 1987.

ON THE OPTIMIZATION OF LARGE EQUI-JOIN QUERIES†

PINTSANG CHANG

Department of Electrical Engineering & Computer Science
Northwestern University
Evanston, Illinois 60208
E-mail: changp@eecs.NWU.Edu

Extended Abstract

The problem of optimizing large (equi-)join queries comes from expert database systems [Krish86] [Swami88]. The equi-join queries may be categorized as *tree queries* and *cyclic queries*. A tree query is a query whose *join query graph* is a tree or there exists an equivalent query whose join query graph is a tree. A cyclic query is a query which is not a tree query. In this paper, we focus on equi-join tree queries. Although the optimization of semi-join scheduling is NP-hard in general [Gouda81] [Yu82], tree queries form an important class of queries such that the complete reduction of irrelevant tuples can be done effectively by semi-joins [Berns81a] [Berns81b].

This paper was motivated by the following:

(i) A concluding remark from [Swami88] that the *iterative improvement* strategy surpasses others because it covers a larger region of the search space.

(ii) In [Krish86], taking a minimum cost spanning tree from a cyclic query graph may lose some join clause(s) or may still result in a cyclic query graph.

(iii) Many of the optimization techniques such as [Sellin79], [Ibara84], [Krish86], [Swami88] consider only the case that a tree query (graph) is obtained, and the optimization is done on this tree query graph; they do not consider the equivalent queries which result from applying the law of transitivity to the query qualification clauses.

(iv) Those who consider equivalent query graphs do not cover the whole set of equivalent queries; e.g., [Yu84] [VanGe86] [Prama88].

We have the next example to illustrate these points.

Example 1: Given a query Q with the qualification q =

(1) R1.A1 = R2.A2 & R2.A2 = R3.A3 &

(2) R2.B2 = R4.B4 & R3.B3 = R4.B4 &

(3) R2.C2 = R3.C3 & R3.C3 = R5.C5 & R4.C4 = R5.C5.

---

† A full-length earlier version manuscript may be found in [Chang88]; same methodology described in this extended abstract has also been applied to the equi-join logic query [Chang89a].

Let 1, 2 and 3 denote three different *equi-join attribute domains* (or briefly, *join domains*) A, B and C respectively. Let V_1 , V_2 and V_3 be the set of referenced relations in the join domains 1, 2 and 3 respectively. V_1 = {R1, R2, R3}, V_2 = {R2, R3, R4} and V_3 = {R2, R3, R4, R5}.

We call V_i the *reference domain* of join domain i.

The join query graph of Q is illustrated in Figure 1. We can see that taking a minimum cost spanning tree (refer to [Krish86],) from the cyclic query graph may lose some join clauses. This flaw holds even if we first take the transitive closures for each reference domain before taking the minimum spanning tree. If we take a (minimum cost) spanning tree for each reference domain, the resulting graph may not be a tree.

By using Yu and Ozsoyoglu's algorithm [Yu79] (also known as *Graham reduction*), we know that Q is a tree query; i.e., an equivalent tree query may be generated. Usually, the generated tree query will be used for further optimization. But, in many cases, there are more than one equivalent tree query. For instance, Figures (a), (b) and (c) in Figure 2 are three distinct but equivalent tree queries of Q. Clearly, the optimal costs of queries in Figure (a), (b) and (c) may not be the same.

/End of example/

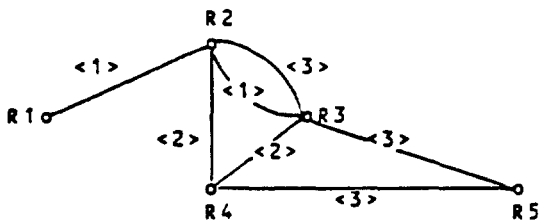


Figure 1: the join query graph of Q in *Example 1*; edges are labeled by the join domains: each edge $R_m \overset{<i>}{\text{o}\!\!-\!\!\text{o}} R_n$ stands for a sigle-attribute join clause.

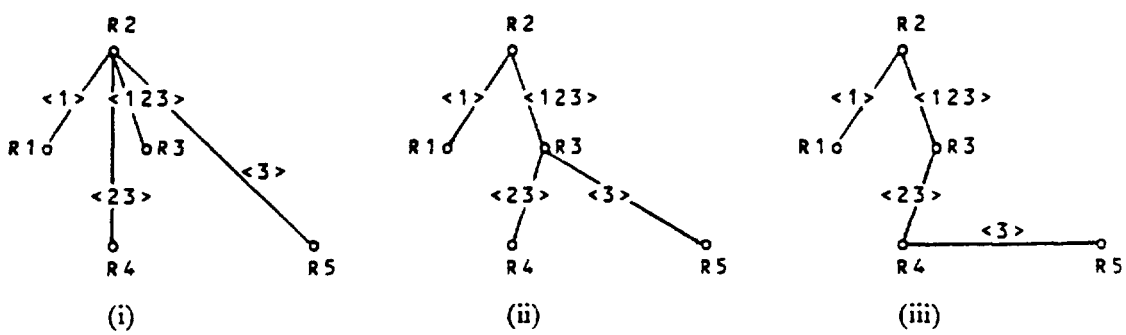


Figure 2: three instances of syntactically equivalent tree queries of Q in *Example 1*; each edge $R_m \overset{<i..j>}{\text{o}\!\!-\!\!\text{o}} R_n$ stands for a multi-attribute join clause.

From the above observations, we argue that the search space of the optimization of query processing should be extended to cover the equivalent queries and may be limited to the *syntactically equivalent* tree queries, given a tree query. Syntactic equivalence of a set of equi-join clauses of the same reference domain is defined by the transitive closure/transitive reduction of the equi-join clauses. The result of transitive closure/transitive reduction of the equi-joins is taking the spanning trees $SP(V_i)$ of the complete graph $J_i^+(V_i)$ formed by the equi-join attribute reference domain V_i. Note that if two queries are of syntactic equivalence, semantic equivalence follows immediately such that the same answer will be generated for any database state.

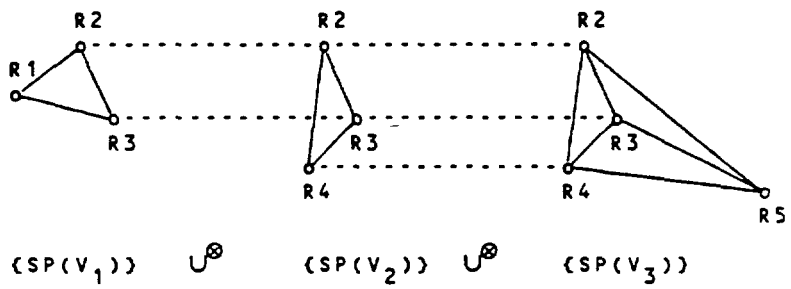


Figure 3: the shuffle unions of $SP(V_i)$ in Q.

Let $\{SP(V_i\}$ be the set of distinct spanning trees of $J_i^+(V_i)$. The set of syntactically equivalent queries of an equi-join query Q is the *shuffle unions* (denoted $\cup^{\otimes}$) of $\{SP(V_i)\}$, for each reference domain V_i; i.e., $\{SP(V_1)\} \cup^{\otimes} .... \cup^{\otimes} \{SP(V_i)\} \cup^{\otimes} .... \cup^{\otimes} \{SP(V_k)\}$ where k is the number of reference domains in Q. Figure 3 illustrates the shuffle unions of the query in *Example 1*. Clearly, generating all the syntactically equivalent queries has a combinatorial explosion (due to taking spanning trees and the combinatoric shuffle) and some of them may not result in a tree query graph. To generate all the syntactically equivalent tree queries, we will employ a graph method called *spanning tree projection map*, denoted $G_p = SP(V_i) \rightarrow G_T$, to analyze the intermediate states of generating syntactically equivalent queries. The search space can be pruned by testing the cyclic condition of $G_p = SP(V_i) \rightarrow G_T$ in linear time. The next theorem states the condition of this cyclic case.

Theorem 1: Let $G_T(V_T, E_T)$ be a tree, $J_i^+(V_i)$ be the transitive closure graph of reference domain i, and $V' = V_i \cap V_T$, $|V'| \geq 2$. The spanning tree projection map $G_p = SP(V_i) \rightarrow G_T$ is cyclic, for all $SP(V_i)$, iff the subgraph of G_T, $G'(V', E')$ where $E' = \{ (R_1, R_2) \mid (R_1, R_2)$ is in E_T and R_1, R_2 are both in $V' \}$, is not a connected component (refer to [Chang89] for a formal proof).

Clearly, the graph connectivity test can be done in linear time with a proper data structure, e.g., the adjacency list. Before we describe the algorithm for generating all the equivalent tree

queries, we first arrange the order of the reference domains to be $< V_1 , ..., V_j , ..., V_k >$ where $|V_j| \leq |V_{j+1}|$, $|V_j \cap V_{j+1}| > 0$, and, if possible, $|V_j \cap V_{j+1}| \geq |V_{j+1} \cap V_{j+2}|$. The reasons for having such an order are two-fold: (i) to limit the branching factor of the state generation tree, and (ii) to guarantee that, for each i-th step of $G_p = SP(V_i) \rightarrow G_T$, G_p will be a connected component. In *Algorithm EQ-TREE-GEN*, we assume that the input $< V_1 , ..., V_k >$ has been ordered as described above.

Algorithm EQ-TREE-GEN: { Generating all the syntactically equivalent tree queries }
Input - reference domains $V_1 , ... , V_k$ of query Q;
Output - all the syntactically equivalent tree queries (ALL-EQ-TREE);
Begin
    G_T := TEMP-EQ-TREE := empty;
    ALL-EQ-TREE := { G_T };
    For each reference domain i do begin
        For each $G_T(V_T , E_T)$ in ALL-EQ-TREE do begin
            V' := $V_T \cap V_i$;
            E' := { (v_1 , v_2) | v_1 , v_2 are in V_i and (v_1 , v_2) is in E_T}
            If G'(V', E') is not a connected component then do begin
                Remove G_T from ALL-EQ-TREE;
                Continue;
            end;
            Else do begin
                For each $SP(V_i)$ s.t. $G_p := SP(V_i) \rightarrow G_T$ is tree do begin
                    Add label i to the edges of $SP(V_i)$ on G_p ;
                    Add G_p to TEMP-EQ-TREE;
                end;
                Remove G_T from ALL-EQ-TREE;
            end;
        end;
        ALL-EQ-TREE := TEMP-EQ-TREE;
    end;
    Output ALL-EQ-TREE;
End;

All the twelve (12) syntactically equivalent tree queries of the Q in *Example 1* generated from *Algorithm EQ-TREE-GEN* are listed in Figure 4. Compared to the shuffle unions which has $3*3*4^2$ alternatives, we may see that *Algorithm EQ-TREE-GEN* incorporating *Theorem 1* as a pruning condi-

tion is indeed effectively generating all the syntactically equivalent tree queryies. Query optimization may be extended to a larger region (i.e., equivalent queries) due to a higher processing speed and employing multiple processors as the hardware technology evolves. Given a tree query, the search space of query optimization may be limited to the set of syntactically equivalent tree queries, under the common assumption that processing tree query graphs is easier than processing cyclic ones. Further research includes applying combinatorial optimization techniques and taking advantage of the common expressions of the syntactically equivalent queries. We expect that further pruning condition(s) for the query optimization will be derived.

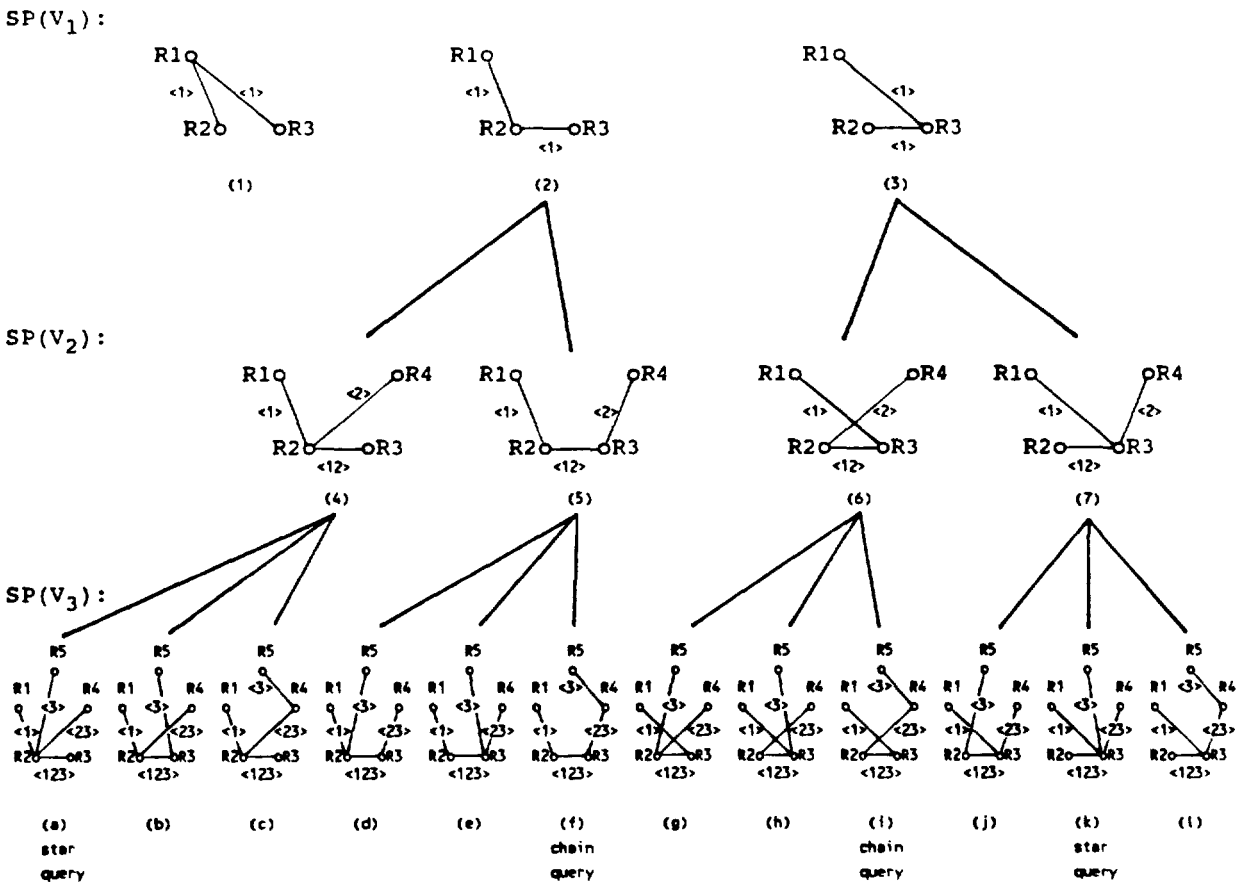


Figure 4: all the twelve syntactically equivalent tree queries of the Q in *Example 1*.

References

[Berns81a] P. A. Bernstein and D-M. Chiu, "Using Semijoins to Solve Relational Queries", *J. ACM*, Vol. 28, No. 1, January 1981, pp.25~40.

[Berns81b] P. A. Bernstein, and N. Goodman, "The Power of Natural Semijoins", *SIAM J. Computing*, Vol. 10, No. 4, December 1981, pp.751~771.

[Chang88] P. Chang, "On Generating Syntactically Equivalent Tree Queries in Distributed Databases" unpublished manuscript, February 1988.

[Chang89] P. Chang, "Query Processing and View Materialization in Logic Database Systems", Ph. D. Dissertation, Northwestern University, June 1989.

[Chang89a] P. Chang and L.J. Henschen, "On Generating Syntactically Equivalent Tree Queries of an Equi-Join Logic Query", submitted for publication.

[Gouda81] M. G. Gouda, "Optimal Semi-Join Schedules for Query Processing in Local Distributed Database Systems", *Proc. ACM-SIGMOD International Conference on Management of Data*, 1981, pp.164~173.

[Ibara84] T. Ibaraki and T. Kameda, "On the Optimal Nesting Order for Computing N-Relational Joins", *ACM Trans. Database Systems*, Vol. 9, No. 3, September 1984, pp.482~502.

[Krish86] R. Krishnamurthy et. al., "Optimization of Nonrecursive Queries", *Proc. VLDB*, August 1986, pp.128~137.

[Prama88] S. Pramanik and D. Vineyard, "Optimizing Join Queries in Distributed Databases", *IEEE Trans. Software Engineering*, Vol. 14, No. 9, September 1988, pp.1319~1326.

[Sellin79] P. Sellinger et. al., "Access Path Selection in a Relational Database Management System", *Proc. ACM-SIGMOD International Conference on Management of Data*, 1979.

[Swami88] A. Swami and A. Goupta, "Optimization of Large Join Queries", *Proc. ACM-SIGMOD International Conference on Management of Data*, Chicago, June 1988, pp.8~17.

[VanGe86] A. Van Gelder, "A Message Passing Framework for Logical Query Evaluation", *Proceedings of ACM-SIGMOD International Conference on Management of Data*, Washington D. C., May 1986, pp.155~165.

[Yu79] C. T. Yu and M. Z. Ozsoyoglu, "An Algorithm for Tree-Query Membership of a Distributed Query", *IEEE Proc. COMPSAC*, 1979, pp.306~312.

[Yu82] C. T. Yu, K. Lam, C. C. Chang and S. K. Chang, "Promising Approach to Distributed Query Processing", *Proc. Berkeley Workshop on Distributed Data Management and Computer Network*, 1982, pp.363~390.

[Yu84] C. T. Yu and C. C. Chang, "Distributed Query Processing", *ACM Computing Surveys*, Vol. 16, No. 4, December 1984, pp.399~433.

Heuristic Search in Query Optimization

Hyuck Yoo and Stéphane Lafortune

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
e-mail: hxy@caen.engin.umich.edu stephane@caen.engin.umich.edu

The importance of query optimization in centralized and distributed relational database systems is widely recognized. One of the key components in query optimizers is the search module. The ability to efficiently find an optimal (or good) solution among all possible alternatives is indeed essential to the performance. In particular, in future application areas of database systems such as Computer Aided Design and Artificial Intelligence, the complexity of query processing is expected to be much greater than in conventional areas so that the efficiency issue becomes more critical.

We are developing new query optimization methods based on heuristic search. Our objective is to achieve search efficiency without relaxing optimality. (Clearly, optimality is subject to the given environment such as cost model and join method.) We employ the A* algorithm, which is probably the best known heuristic search technique. More precisely, we model the steps of query processing in terms of states in a search space, and we develop heuristic information that is needed by the A* algorithm in order to proceed with the search. Unlike many cases that employ "heuristics" to ease the computational efforts during exhaustive search, we try to derive heuristic information that satisfies (i) an "admissibility" condition, thus guaranteeing that an optimal path is found between the initial and final states in the search space, and (ii) a "consistency" condition, in order to avoid redundant calculations during the search [1].

First, we consider the problem of finding optimal semijoin reduction sequences (or programs) for a given tree query. Query optimization strategies based on the reduction of the referenced relations by means of semijoins have received considerable attention. The limitations of such strategies have to do with computational efficiency (very large search space of semijoin reduction sequences), optimality of the solution (when heuristics are used), and generality of the class of queries allowed (e.g., simple queries, chain queries, tree queries). We present a new method that "intelligently" navigates the space of all semijoin sequences and returns an optimal solution.

A database state is defined to be a set of relations. When a semijoin is performed to a relation in a database state, the relation is reduced. The database state is then transformed into a new one, where the relation is replaced by its reduced version. For example, if $x_0 = \{r_1, r_2, r_3\}$ is the initial database state and a semijoin from r_2 to r_1 is applied, then the resulting state is $\{r_1 \propto r_2, r_2, r_3\}$. The initial database state is determined from the given query and represented as a query graph. A sequence of semijoins performed on the given initial database state generates a sequence of transformed database states. The goal state is the state in which all relations are fully reduced. During the optimization, alternative plans are represented by trajectories from the initial state and the best plan is constructed step by step.

The generation of states is controlled by an evaluation function $f(x) = g(x) + h(x)$ on each state x, where $g(x)$ estimates the minimum cost from the initial state to x, and where $h(x)$ estimates the minimum cost from x to the goal state. The function h represents the heuristic information that determines the power of the algorithm. The admissibility condition requires that $h(x) \leq h^*(x)$ for any x, where $h^*(x)$ denotes the optimal cost from x to the goal state, and the consistency condition requires that $h(x_1) \leq g^*(x_1, x_2) + h(x_2)$, where $g^*(x_1, x_2)$ is the minimum cost from x_1 to x_2. Heuristic information based on edges in query graphs is derived through simple observations on semijoins that can be done in the future. We show that the derived h satisfies both conditions.

With the state transformations and the evaluation function, an optimal semijoin program is obtained as follows. 1) Expand the given initial state, i.e., generate its successors each of which represents a transformed state by a semijoin. The number of successors is equal to the number of all possible semijoins in the initial state. 2) Calculate $f(x)$ for each successor x. 3) Choose a state with the smallest f value and expand it. 4) Repeat steps 2 and 3 until the goal state is chosen.

We implemented a prototype in Common Lisp and measured its performance. The experimental results show that our method prunes the state space drastically to find an optimal solution very efficiently. On average, less than five percent of the search space is searched before an optimal solution is found. For star queries, which are known to have large search spaces, an optimal semijoin sequence can be found by searching less than only one percent of the search space. Moreover, we have observed that the improvement increases as the queries get more complex. By measuring the run-time, the overhead of obtaining the heuristic information was shown to be negligible as compared with the time savings that it achieves in the search. Other advantages of the method are: (i) ease of implementation; (ii) generality of the cost model considered; and (iii) ability to handle tree queries with arbitrary target lists. For details, readers may refer to [8].

We also study the optimization of tree join queries. Algorithms which yield an optimal solution for this problem employ either exhaustive search or dynamic

programming. Considering that this approach may not be successful for new future applications with large search spaces, a probabilistic approach was proposed [2,6]. We have considered a heuristic search approach.

Because there are many methods for performing a 2-way join operation, in order to make our method application-independent, we do not assume any specific method for executing a join operation. Instead, we assume that the cost of each join operation is given. Let A and B be two operand relations. Then the cost of executing the join of the two is given as *cost*(A, B), where $cost(\cdot, \cdot)$ is a function that is provided to the search module of the query optimizer. The justification for this assumption is as follows. In [5], it is argued that once the first k relations are joined, the method to join the composite to the (k+1)-st relation is independent of the order of joining the first k. Hence, given two relations, the best method for joining them can be chosen independently of how these relations were obtained. In the context of this assumption, we concentrate on finding the best join order and the best way to choose processing sites in the distributed case in order to achieve the minimum total cost. ([6,4,3] have a similar approach.)

The cost function $cost(\cdot, \cdot)$ is assumed to be monotonic in its arguments. Let s(A) be the size of relation A. Monotonicity means that $s(C) \geq s(A)$ and $s(D) \geq s(B)$ imply $cost(C, D) \geq cost(A, B)$. We believe that this assumption is reasonable because bigger relations will naturally cause the same or more cost. With this assumption, N-way joins are viewed as sequences of 2-way joins. By repeatedly applying 2-way joins to database states, a N-way join can be represented as a sequence of transformed database states. Both nonlinear and linear joins are considered to find a best join order.

For each state, an estimate cost of future joins is calculated and is shown to be a lower bound of the optimal cost. The consistency condition is also proved. Then the repeated expansion of states and the evaluation function lead to an optimal join order. The search efficiency has been tested through experiments. The queries used in the experiments have five to twenty joins. The improvement is from 50% to 75% as compared with dynamic programming. Interestingly, the experiments show different behaviors for linear and nonlinear joins. A complete presentation can be found in [7].

We are currently working on the extension of this heuristic search method to multiple query optimization. While most of the previous work on this problem used given individual access plans of queries to find a global access plan, we construct a global access plan directly from each query graph. A pruning mechanism is also introduced to significantly improve the search by using parallelism.

References

[1] P. E. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[2] Y. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. 1987 ACM-SIGMOD Int. Conf. on Management of Data*, pages 9–22, San Francisco, CA, May 1987.

[3] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. 12th Int. Conf. on Very Large Data Bases*, pages 128–137, Kyoto, August 1986.

[4] S. Lafortune and E. Wong. A state transition model for distributed query processing. *ACM Trans. Database Systems*, 11(3):294–322, September 1986.

[5] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in relational database management systems. Technical Report RJ 2429, IBM Research Laboratory, San Jose, January 1979.

[6] A. Swami and A. Gupta. Optimizing large join queries. In *Proc. 1988 ACM-SIGMOD Int. Conf. on Management of Data*, pages 8–17, Chicago, June 1988.

[7] H. Yoo. *Intelligent Search in Query Optimization.* PhD thesis, University of Michigan, 1989. In preparation.

[8] H. Yoo and S. Lafortune. An intelligent search method for query optimization by semijoins. *To appear in IEEE Trans. Knowledge and Data Engineering*, 1989.

Multiple Query Optimization: To Use or Not To Use?

Sharma Chakravarthy
Xerox Advanced Information Technology
Four Cambridge Center, Cambridge, MA 02142.
Email: sharma@xait.xerox.com

Introduction Multiple query optimization (MQO) is the process of simultaneously optimizing two or more queries related according to some criteria. Two pertinent questions that need to be answered in this context are: i) How much do we gain by processing n queries simultaneously? and ii) What is the the cost of optimizing the set as a whole in comparison to the cost of optimizing each query separately?. There is a definite known upper bound on the gain when a set of n queries are processed simultaneously. However, if the cost of optimizing and evaluating the set as a whole is less than the sum of the costs of optimizing each query individually, then it is beneficial to use MQO techniques.

The purpose of this paper is to make a case for multiple query optimization as a useful technique for processing queries in databases currently being developed for new classes of applications. The overall objectives of this paper are: to create an awareness of this promising area of research and more importantly, to spark a vigorous discussion of the relevance and applicability of this area to cull out topics for further research. In the remainder of this paper, we will introduce the notion of advanced databases, present scenarios that can benefit from multiple query optimization techniques, state the MQO problem concisely, and identify key topics for further investigation.

Advanced DBMSs The importance of databases in non-traditional applications (e.g., engineering design, stock market applications, CASE, and document management) is well understood and needs no elaboration. Conventional DBMSs do not adequately meet the requirements of non-traditional applications. In order to support these applications, the functionality of conventional DBMSs is being extended in several ways. At least three types of DBMSs (termed advanced DBMSs in the rest of this paper), which overlap in their functionality, can be identified readily:

1. *Object-oriented database systems:* to support new object classes, their operations and complex objects in a modular and extensible manner.
2. *Expert database systems:* to integrate large rule bases and their processing with relational database systems, and
3. *Active database systems:* to support situation monitoring, timing constraints, and a host of related features.

Several research prototypes that are being pursued actively, such as POSTGRES [STON86], EXODUS [CARE86, CARE87], PROBE [DAYA87], DIPS [SELL88b], and HiPAC [DAYA88a, DAYA88b] can be viewed as advanced database systems as they fall into one or more of the categories described above. Realization of these DBMSs requires extensions to conventional query-processing techniques, as well as reevaluation of the architecture and functionality of existing query processors.

Current query processors optimize queries *one at a time*. Significant performance improvements can be achieved by grouping several queries and optimizing the group as a whole. In fact, the need for processing groups of queries arises naturally in many non-traditional applications as explained below.

Application Scenarios We identify a variety of application scenarios that require the capabilities of advanced DBMSs and in each case indicate opportunities for using MQO techniques.

1. *Situation Monitoring*: An essential characteristic of an active database is the capability to monitor situations defined over the state of the database. Situations can be expressed using the data manipulation language (as in POSTGRES and SYBASE), using rules (as in HiPAC), or using rules with matching patterns (as in DIPS). HiPAC rules are of the form <event, condition, action>, where condition may reference data in the event and in the database. In DIPS, matching patterns of rules are used to describe data that can trigger the execution of actions.

   The set of all conditions forms a potentially large set of predefined queries that have to be evaluated efficiently. In HiPAC [ROSE89] and DIPS, an event may trigger the evaluation of one or more conditions, leading to the optimization of a set of conditions rather than their evaluation one at a time.

2. *Support for Application-Oriented Objects*: Support for complex objects requires a capability for specifying the objects in the user's mental model and operations on the objects. Internally, a user object may be represented as sets of database views. Evaluation of these views is likely to generate overlapping queries, requiring MQO techniques.

   For example, consider a transaction in which a documentation group downloads specifications and drawings for a product. Suppose that in the stored database, Documents and Drawings are associated with Parts, and a Product contains many Parts. To make things more manageable, views Prod_Specs(Prod#, Spec#, Author) and Prod_Drawings(Prod#, Draw#, Artist) have been defined:

   View Prod_Specs = [Prod_Part(Part#,Prod#) $\text{join}_{\text{Part\#}}$ Specs(Doc#,Author,Part#)]

   The definition of Prod_Drawings is similar. Now the download transaction for product 1234 is:

   [Select {Prod_Specs| Prod#= 1234}; Select {Prod_Drawings| Prod#= 1234}]

   When expanded, the two views represent queries with very substantial overlap.

   In this example, there is no natural way to combine the two views into one relation. Joining them yields a cartesian product of Specs and Drawings for each Part.

3. *Expert and Deductive Databases*: View definitions composed of union-compatible expressions (or equivalently intensionally defined predicates in expert database systems) are likely to have substantial overlap. Evaluation of a query on such views (or predicates) can benefit from multiple query-evaluation techniques by exploiting intra-query commonalities.

   As an example, the query Select {Employed | nationality <> US} has substantial overlap when the following definitions are used.

   Base relations: Personnel(id, name, nationality)
   Faculty(id, rank, compensation)
   Staff(id, rank, compensation)

   View: Employed = (Faculty join_{id} Personnel) union (Staff join_{id} Personnel)

4. *Semantic Query Optimization*: Semantic query optimization aims to use application semantics (e.g., integrity constraints) to optimize queries. This approach essentially introduces a level of abstraction in which several semantically equivalent queries are generated using residues that are derived from integrity constraints. Specifically, the process of residue generation [CHAK85, LEE88a] analyzes several integrity constraints, which can be mapped to the problem of multiple query processing.

5. *Distributed/Multi-Database Query Processing*: In a distributed (or even a multi-database environment), queries are decomposed, and query fragments are directed to particular sites (or databases) for processing. Distribution of the database reduces the size of the data stored at each node, thereby increasing the locality of reference for the queries processed at a given node. Replicated databases provide an additional opportunity — that of choosing the site at which a subquery is sent for processing to increase the probability of overlap with other subqueries. Hence, queries processed at a site may have a lot of overlap of the data they access. MQO techniques can be used to exploit inter-query commonalities in a distributed DBMS. In both distributed and multi-database environments batching of queries is likely to be one of the query-processing strategies that can certainly benefit from MQO techniques. Preliminary work in this area has been reported in [SU86, PARK87].

It is evident from the above that the need for the use of multiple query processing techniques arises naturally in various applications for which advanced DBMSs are being developed. It is likely to become an extremely important component of future query optimizers, hence needs to be developed from that perspective.

Problem Statement The idea behind MQO is to minimize the cost of evaluation of a set of queries related according to some criteria by evaluating them simultaneously. Characteristics of queries that can be exploited for minimizing the total cost of evaluation include identical subexpressions or even subexpressions that subsume one another, common scan of relations, using cached results from earlier computation, etc. The *multiple-query optimization problem* is to find an execution plan (termed *multi-strategy*) for a set of queries that minimizes the total computational cost, usually exploiting overlaps among queries in the group.

Formally, let $Q = \{Q_1, Q_2, \ldots, Q_n\}$ be a set of queries to be optimized as a unit. Without loss of generality, assume that n answer sets have to be computed for Q. Let $MQS(Q) = \{mqs_1(Q), mqs_2(Q), \ldots, mqs_m(Q)\}$ be the set of all multi-query strategies for the set of queries Q. The problem of multiple-query optimization is to select a multi-strategy $mqso(Q)$ such that

$$\text{Cost(mqso(Q))} = \text{MIN } \{ \text{ Cost(mqs)} \mid \text{mqs belongs to MQS(Q)}\}$$

where $mqso(Q)$ is an optimal multi-strategy.

Research Topics Solution to the problem of multiple query optimization requires further research in the following areas:

Generation of the strategy space: Even for the optimization of a single query, it is impractical to generate the entire strategy space exhaustively. In the case of multiple queries, it is even more important that the entire strategy space is not generated. Furthermore, generation of strategies for each query in the set and then combining them does not seem to be a viable strategy. Techniques for generating a small number of promising multi-strategies need to be developed. Overlap of queries need to be used as a constraint for restricting the strategy space generated. In addition, heuristics for restricting the strategy space as well as for choosing promising candidate strategies need to be developed.

Organization and representation of the strategy space: The likelihood of a large number of multi-

strategies (the number of multi-strategies increase multiplicatively with the number of strategies for individual queries) strongly suggests that the strategy space be organized intelligently. This has to be considered in conjunction with the representation chosen as well as the techniques used for searching the strategy space. Partitioning of the strategy space into two or more spaces each exploring different sets of strategies seems to be useful [CHAK88]. An efficient way to represent the strategy space that minimizes redundancy is required. The representation chosen needs to accommodate: i) overlap of computations within a multi-strategy and ii) overlap among distinct multi-strategies. As suggested in [ROSE88], an AND/OR graph seems to be a good choice.

Constructs for expressing multi-strategies: Unlike an operator tree whose operator nodes take one or more inputs and generate a *single* output, operators in multi-strategies are likely to produce multiple outputs (e.g., join of a relation with two distinct relations). Even though this can be expressed using a conventional operator tree constructs for expressing multi-strategies will not only reduce the size of the strategy space, but also contribute towards the expansion of an operator node from the logical level to the physical level.

Cost models and search on the multi-strategy space: Existing cost models seem to be adequate and can be used for computing the cost of a multi-strategy. However, the search of an AND/OR graph for a minimum cost strategy (solution graph) is NP-Complete. Hence heuristic algorithms need to be extended/developed for searching the space of multi-strategies efficiently.

Common subexpression identification: The underlying premise for multiple query optimization is that there is substantial overlap among queries. This overlap needs to be identified and exploited. There has been some work [JARK84, SELL88a, CHAK86, ROSE88, CHAK88] on the identification of common sub-expressions but it needs to be extended further.

Architecture of a multiple query optimizer: A multiple query optimizer can be viewed as a generalization of a single query optimizer. In [ROSE88], we identified the subproblems of a multiple query optimizer and their organizational structure to produce a modular multiple query optimizer. The ultimate objective is to design a modular query optimizer in which a variety of techniques can be mixed and matched efficiently based upon application requirements. Results from current research on extensible query optimizers [BATO87, FREY87, GRAE87, LOHM87, LEE88b] have been encouraging and seem to be useful for the design of a multiple query optimizer.

References

[BATO87] D. Batory. A Molecular Database Systems Technology. Computer Science Department TR-87-23. University of Texas at Austin.

[CARE86] M. Carey, et al., The Architecture of the EXODUS Extensible DBMS. *Proc. Int'l. Workshop on Object-Oriented database Systems*. CA. Sept. 86.

[CARE87] M. Carey and D. DeWitt. An Overview of the EXODUS Project. *Database Engineering*. June 1987.

[CHAK85] U. S. Chakravarthy. Semantic Query Optimization in Deductive Databases. Ph.D. Thesis. Univ. of Maryland. College Park. 1985.

[CHAK86] U. S. Chakravarthy and J. Minker. Multiple Query Processing in Deductive Databases using Query Graphs. *Proc. of 12-th VLDB Conf.*, Kyoto. Japan. August 86.

[CHAK88] U. S. Chakravarthy., Multiple Query Optimization: Organization of the Strategy Space and the Generation of Shared Multi-Strategies. Submitted for publication.

[DAYA87] U. Dayal et al., PROBE Final Report, *XAIT Technical Report XAIT-87-02*, 1987.

[DAYA88a] Dayal, U., et al., HiPAC: A Research Project in Active, Time-Constrained Database Management, XAIT Interim Report XAIT-88-02, Computer Corporation of America, 1988.

[DAYA88b] U. Dayal, Active Database management Systems, *Proc. of Conf. of Data and Knowledge Bases*, Jerusalem, 1988.

[DAYA88c] U. Dayal, A. Buchmann, and D. McCarthy, Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System,

[FREY87] J. C. Freytag, A Rule-Based View of Query Optimization, *Proc. ACM-SIGMOD Conference on Management of Data*, San Francisco, 1987.

[GRAE87] G. Graefe and D. DeWitt, The Exodus Optimizer Generator, *Proc. ACM-SIGMOD Conference on Management of Data*, San Francisco, 1987.

[JARK84] M. Jarke, Common Subexpression Isolation in Multiple Query Optimization, *In Query Processing in Database Systems, (W. Kim, D. Reiner, and D. Batory, Eds.)*, Springer-Verlag, 1984.

[LEE88a] S. Lee and J. Han, Semantic Query Optimization in Recursive Databases, *Proc. of 4th International Conf. on Data Engineering*, pp. 444-451, 1988.

[LEE88b] M. K. Lee, J. C. Freytag, and G. M. Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *Proc. of 14th Int'l Conf. on Very Large Databases*, Long Beach, pp. 218-229, 1988.

[LOHM87] G. Lohman, Grammar-like Functional Rules for Representing Query Optimization Alternatives, *Proc. of ACM-SIGMOD*, 1988, pp. 18-27.

[PARK87] J. T. Park and T. J. Torey, A Knowledge-based Approach to Multiple Query Processing in Distributed Database Systems, *Proc. of ACM-IEEE Fall Joint Computer Conf.*, Dallas, TX, October 1987.

[ROSE88] A. Rosenthal and U. S. Chakravarthy, Anatomy of a Modular Multiple Query Optimizer, *Proc. of 14th Int'l Conf. on Very Large Databases*, Long Beach, 1988.

[ROSE89] A. Rosenthal, U. S. Chakravarthy, B. Blaustein, and J. Blakeley, Situation Monitoring in Active Databases, To appear in the *Proc. of 15th Int'l Conf. on Very Large Databases*, Amsterdam, 1989.

[SELL88a] T. K. Sellis, Multiple-Query Optimization, *ACM TODS*, Vol. 13, No. 1, 1988.

[SELL88b] T. K. Sellis, C-C. Lin, and L. Raschid, Using Relational DBMSs for Data Intensive Production Systems: The DIPS System, *Proc. of the AAAI-88 Workshop on Databases in Large AI Systems*, St. Paul, MN, August 1988.

[STON86] M. Stonebraker and L. Rowe, The Design of POSTGRES, *Proc of ACM-SIGMOD*, 1986.

[SU86] S. Y. Su, et al., A Distributed Query Processing Strategy Using Decomposition, Pipelining and Intermediate Result Sharing, *Proc. IEEE Conf on Data Engineering*, Los Angeles, CA, February 1986.

QUERY OPTIMIZATION IN TEMPORAL DATABASES

Arie Segev and Himawan Gunadhi

School of Business Administration
University of California
and
Computer Science Research Department
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, CA 94720
e-mail: segev@csam.lbl.gov and gunadhi@csam.lbl.gov

ABSTRACT. The importance of temporal data models is in their ability to capture the complexities of real world phenomena which are inherently dependent on time. Traditional approaches, such as the relational model of data, are incapable of handling all the nuances of such phenomena. Temporal models open up the possibility for new types of operations that enhance the retrieval power of a DBMS. One of the potential drawbacks of such models is the potential for processing inefficiency: the large size of stored data for many applications, and the complexity of time-oriented queries may yield unsatisfactory performance. Consequently, it is very important to devise efficient query optimization strategies.

1. RELATIONAL REPRESENTATION OF TEMPORAL DATA

A convenient way to look at temporal data is through the concepts of *Time Sequences* (*TS*) and *Time Sequence Collection* (*TSC*) [13]. A *TS* represents a history of a temporal attribute(s) associated with a particular instance of an entity or a relationship. The entity or the relationship is identified by a surrogate (or equivalently, the *time-invariant key* [9]). For example, the salary history of employee #1 is a *TS*. A *TS* is characterized by several properties, such as the time granularity, lifespan, type, and interpolation rule to derive data values for non-stored time points. In this abstract, we are concerned with two types -- *stepwise constant* and *discrete*. Stepwise constant (*SWC*) data represents a state variable whose values are determined by events and remain the same between events; the salary attribute represents *SWC* data. Discrete data represents an attribute of the event itself, e.g. number of items sold. Time sequences of the same surrogate and attribute types can be grouped into a time sequence collection (*TSC*), e.g. the salary history of all employees forms a *TSC*.

There are various ways to represent temporal data in the relational model; detailed discussion can be found in [14]. In our work, we assume first normal form relations (1NF). Table 1 shows two ways of representing *SWC* data. The representations can be different at each level (external, conceptual, physical), but we are concerned with the tuple representation at the physical level. The representation in Table 1(b) stores data only for event points and requires explicit storage of *null* values to indicate the transition of the state variable into a non-existence state. Also, the tuples should be ordered by time in order to determine the values between two consecutive event points. Both representations require the use of the lifespan metadata; it is required for the time-interval representation since we do not store non-existence nulls explicitly, for example, the lifespan is needed in order to correctly answer the query "what was the commission rate of E2 at time 12?". In order to generalize the analysis, we assume *SWC* data using the time-interval representation; for discrete data, using time-intervals is superfluous since the start time T_S is equal to the end time T_E for each tuple. We will point to cases where simplified algorithms can be used when we describe the event-join operation. We use the terms *surrogate*, *temporal attribute*, and *time attribute* when referring to attributes of a relation. For example, in Table 1, the surrogate of the MANAGER relation is E#, MGR is a

This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

MANAGER	E#	MGR	T_S	T_E
	E1	TOM	1	5
	E1	MARK	9	12
	E1	JAY	13	20
	E2	RON	1	18
	E3	RON	1	20

COMMISSION	E#	C_RATE	T_S	T_E
	E1	10%	2	7
	E1	12%	8	20
	E2	8%	2	7
	E2	10%	8	20

(a) time-interval representation

MANAGER	E#	MGR	T
	E1	TOM	1
	E1	∅	6
	E1	MARK	9
	E1	JAY	13
	E2	RON	1
	E2	∅	19
	E3	RON	1

COMMISSION	E#	C_RATE	T
	E1	∅	1
	E1	10%	2
	E1	12%	8
	E2	∅	1
	E2	8%	2
	E2	10%	8

(b) time-point representation

Table 1: Representing Step-Wise Constant Data with Lifespan = [1, 20]

temporal attribute, and T_S and T_E are time attributes. We refer to the data construct as a 'relation', but we mean a 'temporal relation'; it is different from a standard relation because of the associated meta-data. We assume that all relations are in first temporal normal form (1TNF) [14]. 1TNF requires that for each combination of surrogate instance, time point in the lifespan, and temporal attribute (or attributes) there is at most one temporal value (or a unique combination of temporal values). Note that 1NF does not imply 1TNF, for example, the relation COMMISSION in Table 1(a) would not be in 1TNF if for any surrogate instance there were two tuples with the same commission rate value and intersecting time intervals.

2. EVENT JOINS

There are many temporal operators tha require optimization, but the join types [2,6] require the most attention. In this section we provide more details about one of these operators -- the *Event-Join*, introduced in [14]. An event-join groups several temporal attributes of an entity into a single relation. This operation is extremely important because due to normalization, temporal attributes are likely to reside in separate relations. To illustrate this point, consider an employee relation in a conventional database. If the database is normalized we are likely to find all the attributes of the employee entity in a single relation. If we now define a subset of the attributes to be temporal (e.g., salary, job-code, manager, commission-rate, etc.) and they are stored in a single relation, a tuple will be created whenever an event affects at least one of those attributes. Consequently, grouping temporal attributes into a single relation should be done if their event points are synchronized. Regardless of the nature of temporal attributes, however, a physical database design may lead to storing the temporal attributes of a given entity in several relations. The analogy in a conventional database is that the database designer may create 3NF tables, but obviously, the user is allowed to join them and create an unnormalized result.

Let $r_i(R_i)$ be a relation on scheme $R_i = \{S_i, A_{i1}, ..., A_{im}, T_S, T_E\}$. In many instances we illustrate the concepts using a single temporal attribute, that is, $m = 1$; all apply to any $m > 1$. Also, when the two surrogate *types*

S_i of R_i and S_j of R_j are the same, we simply use S. Instances of surrogate S are denoted by $s1, s2, \cdots$. We use x_i to refer to an arbitrary tuple of r_i; $x_i(A)$ is the value of attribute A in tuple x_i. In order to describe the event-join between r_1 and r_2, we first present two basic operations *TE–JOIN* and *TE–OUTERJOIN*. TE-JOIN is the temporal equivalent of a standard equijoin; two tuples $x_1 \in r_1$ and $x_2 \in r_2$ are concatenated † if their join attribute's values are equal and the intersection of their time intervals is non-empty; the T_S and T_E of the result tuple correspond to the intersection interval. Semantically, this join condition is "where the join values are equal at the same time". Optimization issues in executing general TE-JOINs are discussed in [6]. In the case of event-joins, we are concerned only with a special case of TE-JOINs where the joining attribute is the surrogate. A TE-OUTERJOIN is a directional operation from r_1 to r_2 (or vice versa). For a given tuple $x_1 \in r_1$, outerjoin tuples are generated for all points $t \in [x_1(T_S), x_1(T_E)]$ where there does not exist $x_2 \in r_2$ such that $x_2(S) = x_1(S)$ and $t \in [x_2(T_S), x_2(T_E)]$. Note that all consecutive points t that satisfy the above condition generate a single outerjoin tuple. Using those operations the event-join is done as follows.

r_1 EVENT-JOIN r_2:
temp1 $\leftarrow$ r_1 TE-JOIN r_2 on S
temp2 $\leftarrow$ r_1 TE-OUTERJOIN r_2 on S
temp3 $\leftarrow$ r_2 TE-OUTERJOIN r_1 on S
result $\leftarrow$ temp1 $\cup$ temp2 $\cup$ temp3

The above operations are illustrated in the example of Table 2, where an event-join is performed between the MANAGER and COMMISSION relations of Table 1.

The most troublesome components of the event-join are the outer-joins. The situation is further complicated by the time interval predicate associated with the TE-outerjoin, preventing the usage of non-temporal outerjoin procedures [4,10]. An easy solution that comes to mind is to store all non-existence tuples explicitly, e.g., tuples like (1, ∅, 6, 8) are added to the MANAGER relation of Table 1. In that case the outerjoin components disappear, and the problem reduces to a TE-JOIN on S. Unfortunately, there are many situations where such a 'fix' will degrade overall performance rather than improve it. For example, if the whole S_i domain is represented in relation r_i, representing all non-existence data explicitly will in the worst case double the size of the table (this is the case of alternating state transitions between existence and non-existence). A much worse problem may arise when a relation contains only a fraction of the S-domain values, e.g., if on the average, only 5% of the employees of a large corporation earn commissions, adding to the non-existence data for the 95% other employees to the commission relation will add to storage cost, querying cost (including event joins), and maintenance of the commission relation and any of its associated secondary indexes.

Consequently, we divide event-joins into two types -- 'easy' and 'difficult'. Easy cases are those where the relations contain explicit tuples for all non-existence data and are sorted by (S, T_S). Other cases are regarded difficult, and we are mostly concerned with them. More details about the optimization of event-joins are given in [12]. It should be noted that some of the problems in optimizing event-joins are common to TE-JOIN optimization.

3. DISCUSSION ITEMS

The following important issues have to be addressed:

Architecture of Query Processor

The first issue is whether or not to use a conventional query optimizer for the processing of temporal queries. An implementation such as that of [16] is based on the construction of a temporal database on top of a conventional one. Minimal modification of the underlying processor is likely to cause inefficiencies in the processing of many temporal operators. Since there are operators which are not part of a relational system, An interface has to be designed such that a temporal query is mapped into one or more relational queries, where the latter retrieve a superset of the required data which is then further manipulated by the interface software. The other option is to augment

† It is not a standard concatenation since only one pair of T_S and T_E is part of the result tuples.

MANAGER TE-JOIN COMMISSION ON E#

temp1	E#	MGR	C_RATE	T_S	T_E
	E1	TOM	10%	2	5
	E1	MARK	12%	9	12
	E1	JAY	12%	13	20
	E2	RON	8%	2	7
	E2	RON	10%	8	18

MANAGER TE-OUTERJOIN COMMISSION ON E#

temp2	E#	MGR	C_RATE	T_S	T_E
	E1	TOM	Ø	1	1
	E2	RON	Ø	1	1
	E3	RON	Ø	1	20

COMMISSION TE-OUTERJOIN MANAGER ON E#

temp3	E#	MGR	C_RATE	T_S	T_E
	E1	Ø	10%	6	7
	E1	Ø	12%	8	8
	E2	Ø	10%	19	20

MANAGER EVENT-JOIN COMMISSION

result	E#	MGR	C_RATE	T_S	T_E
	E1	TOM	Ø	1	1
	E1	TOM	10%	2	5
	E1	Ø	10%	6	7
	E1	Ø	12%	8	8
	E1	MARK	12%	9	12
	E1	JAY	12%	13	20
	E2	RON	Ø	1	1
	E2	RON	8%	2	7
	E2	RON	10%	8	18
	E2	Ø	10%	19	20
	E3	RON	Ø	1	20

Table 2: Event-Join Derivation

a relational optimizer by temporal modules, while the extreme case is to design a temporal database from scratch. (Are extensible DBMSs which are designed today sufficient to support general temporal capabilities?).

Organization and Indexing of Data

Conventional indexing techniques are not sufficient for efficient temporal data retrieval. If appropriate indexing structures are available, it can improve performance substantially. Associated with indexing is the physical organization of data, i.e. whether it is clustered, sorted on one or more keys, or physically kept in contiguous or near contiguous block order. Different approaches have been undertaken, for example, reverse chaining of history tuples [8], multidimensional partitioning for static data [11], R-trees and versioning [3], multistore storage systems [1], and nested indexes [5, 7]. Research in this area has focused on single relation operations, but there is the potential for performance gains if multirelational indexing are developed. We have also paid attention to Append-only database which are attractive in some cases.

Statistical Data

Selectivity measures are fairly established for standard relational databases. There are several major distinctions between temporal and non-temporal relations. First, there is a time dependency among tuples belonging to the same surrogate. These tuples are orderable according to the T_S or T_E time-stamp. Second, the generation of a new tuple for a surrogate is determined by the first change among the temporal attributes associated with it: knowing the rate at which each attribute changes will provide information about the distribution of attributes over time. For example, if we know that historically employees remain with the company for 5 years and 100 new hirings are made annually, we can determine something about the distribution of tuples across time.

The maintenance and availability of statistical information about the relation is a critical aspect of temporal query processing. Metadata may include information on the life-span of the relation and characteristics of its time attributes; and the probability distribution of each temporal attribute. While such data may be costly, they are useful in capturing information that can influence selectivity measurement. Note also that even if no index is maintained on an attribute, it may still be desirable to keep detailed statistics about it.

Time Representation

We are concentrating on the case of a single time line representation of data. In [15] the idea of multiple time lines was introduced. Clearly such a model will make the problems of indexing and query processing much more difficult.

REFERENCES (for more references on temporal databases, see the special issue of IEEE Data Engineering; vol. 11, No. 4, Dec. 1988).

[1] Ahn, I., Towards an Implementation of Database Management Systems with Temporal Support, *Proceedings of the International Conference on Data Engineering*, February 1986, pp. 374-381.

[2] Clifford, J., Croker, A., The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans, *Proceedings of the International Conference on Data Engineering*, February 1987, pp. 528-537.

[3] Colovson, C.P., Stonebraker, M., Indexing Techniques for Historical Databases, *Proceedings of the International Conference on Data Engineering*, February 1989.

[4] Dayal, U., Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers, *Proceedings of the International Conference on Very Large Data Bases*, August 1987, pp. 197-208.

[5] Gunadhi, H., Segev, A., Physical Design of Temporal Databases, Lawrence Berkeley Lab Technical Report LBL-24578, December 1988.

[6] Gunadhi, H., Segev, A., A Framework for Query Optimization in Temporal Databases, Lawrence Berkeley Lab Technical Report LBL-26417, December 1989.

[7] Gunadhi, H., Segev, A., Indexing Structures for Temporal Databases, In preparation.

[8] Lum, V., Dadam, P., Erbe, R., Guenauer, J., Pistor, P., Walch, G., Werner, H., Woodfill, J., Designing DBMS Support for the Temporal Dimension, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1984, pp. 115-130.

[9] Navathe, S., Ahmed, R., A Temporal Relational Model and a Query Language, UF-CIS Technical Report TR-85-16, Univ of Florida, April 1986.

[10] Rosenthal, A., Reiner, D., Extending the Algebraic Framework of Query Processing to Handle Outerjoins *Proceedings of the International Conference on Very Large Data Bases*, August 1984, pp. 334-343.

[11] Rotem, D., Segev, A., Physical Organization of Temporal Data, *Proceedings of the International Conference on Data Engineering*, February 1987, pp. 547-553.

[12] Segev, A., Gunadhi, H., Event-Join Optimization in Temporal Relational Databases, *Proceedings of the International Conference on Very Large Data Bases*, August 1989, forthcoming. Also Lawrence Berkeley Lab Technical Report LBL-26600, January 1989.

[13] Segev, A., Shoshani, A., Logical Modeling of Temporal Databases, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1987, pp. 454-466.

[14] Segev, A., and Shoshani, A., The Representation of a Temporal Data Model in the Relational Environment, *Lecture Notes in Computer Science*, Vol 339, M. Rafanelli, J.C. Klensin, and P. Svensson (eds.), Springer-Verlag, 1988, pp. 39-61.

[15] Snodgrass, R., Ahn, I., A Taxonomy of Time in Databases, *Proceedings of ACM SIGMOD International Conference on Management of Data* , May 1985, pp. 236-246.

[16] Snodgrass, R., Ahn, I., Performance Analysis of Temporal Queries, TempIS Document No.17, Department of Computer Science, University of North Carolina, Chapel Hill, August 1987.

Optimization of Alerters/Triggers/Rules in Active DBMSs†

Sharma Chakravarthy‡
Xerox Advanced Information Technology
Four Cambridge Center, Cambridge, MA 02142.
Email: sharma@xait.xerox.com

Introduction In this paper we articulate the problem of efficient rule management and its evaluation to meet the requirements of an active database management system. We first analyze the characteristics of this problem and contrast them with that of processing queries in a conventional, passive DBMS. Based on this analysis, we identify techniques that are especially useful for managing and evaluating rules efficiently. Finally, we present the architecture of a rule processor along with its functional components and its interaction with other components of an active DBMS

An *active database management system* is characterized by its ability to monitor and react to several types of events – both database and non-database — in a timely and efficient manner There have been several attempts [STON85, STON87, DARN87, DITT86] at integrating this capability with conventional, passive DBMSs In fact, this capability has been shown to be pivotal for supporting a variety of database functions (e.g., integrity constraint enforcement, materialization and maintenance of derived data, access/authorization control) in an elegant manner.

Providing active capability requires several conceptual as well as architectural extensions to a conventional DBMS One such extension — perhaps a primary one— is the component that is responsible for the efficient management and evaluation of triggers and alerters (*RuMES* for Rule Management and Evaluation Subsystem) This article draws upon the work done on situation monitoring [ROSE89] in HiPAC a research project on active, time-constrained database management system being investigated at XAIT [DAYA88a, DAYA88b]. In the rest of the discussion, rules denote HiPAC rules consisting of event, condition, action, and associated context information. These rules have been shown to subsume the functionality of triggers and alerters [DAYA88c].

Characteristics of rules in active DBMSs Rules that are specified to an active database, are temporally persistent. In other words, compared to queries rules have a longer life-span and as a result are likely to be evaluated many times. The set of all rules form a potentially large set of predefined queries that have to be evaluated efficiently. Rules (or its components) may have priorities as well as timing requirements associated with their execution.

The semantics of rule execution (i.e., the execution of its components) is also different from that of query evaluation. Evaluating a rule either as part of the triggering transaction (immediate evaluation) or prior to the commit of the transaction (deferred evaluation) are restrictive in at least two ways: i) they do not capture the semantics of many applications and ii) enforcement of serializability

†This work was supported by the Defense Advanced Research Projects Agency and by Rome Air Development Center under contract No. F30602-87-C-0029. The views and conclusions contained in this paper are those of the authors and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Rome Air Development Center, or the U.S. Government.

‡This paper draws upon the research done as part of the situation monitoring task of the HiPAC project. Besides the author, Barbara Blaustein and Arnie Rosenthal were co-responsible for the situation monitoring task. Other members of the HiPAC project team are: Alex Buchmann, Umeshwar Dayal (currently with DEC), Meichun Hsu, Rivka Ladin, Dennis McCarthy of XAIT, and Michael Carey, Miron Livny, and Rajeev Jauhari of the University of Wisconsin, Madison

severely limits the concurrent evaluation of rules. For example, in an inventory control application, there is no apparent reason to execute the reorder action within the transaction that reduced the quantity on hand below the threshold level. Allowing actions (and even conditions) to occur in separate transactions permits the triggering transactions to finish more quickly and thereby release the system resources earlier and improve transaction response times. Execution of rules relative to transaction boundaries has an impact on the grouping as well as optimization of rules and their components.

The collection of rules needs to be managed efficiently by the system. Operations that need to be supported for rule management include add (or create), delete, enable, and disable. It is not the case that all the rules will be enabled at the same time. Rule sets need to be enabled and disabled as required (dynamically specified as an action of some rule). Rules may also have to be enabled and disabled selectively requiring modifications to optimized versions. Also, there may be rules that are generated by the system itself (e.g., for materializing an intermediate relation) which also need to be enabled/disabled in a consistent manner with the rules it supports.

Optimization issues The techniques that are useful for the optimization of rules can be inferred from the characteristics discussed above. Even though, temporal persistence of rules clearly indicates the benefit of grouping rules and the need for *multiple query optimization* techniques several additional factors need to be taken into account to assure correctness of execution i) coupling of rules (or its components) with the triggering transaction, ii) priority specification, if any of rules and iii) timing constraints, if any, on the evaluation of rules. In addition, repeated nature of evaluation of rules indicate the need for *materialization of intermediate results* as well as *incremental evaluation* of rules where possible. Incremental evaluation strategy is based on the properties of the operators in the condition and the data resulting from events This strategy will reduce the computation and in some cases may even eliminate the computation itself Timing constraints may necessitate exhaustive optimization and the use of main memory processing techniques. Furthermore, the need for the management of rules influence the way in which the optimized versions of the rules are maintained. The representation of optimized versions of rules need to be amenable to *incremental modifications* rather than requiring recompilation/reoptimization Finally, the properties of the components of rules along with how the rules are to be executed relative to the transaction that triggered them can be exploited for performing optimization For example, if the condition and action are queries (i.e., no side-effects) and the rules are to be executed as part of the triggering transaction, then multiple query optimization may be relevant.

Architectural issues The architecture of RuMES has to take into account its interaction with other components of an active DBMS From the point of view of extensibility the interfaces need to be well defined. Evaluation of rules is governed by the occurrence of one or more events specified as part of the rule and their detection As a result, RuMES needs to interface with a variety of event detectors (database event detector -- one which detects events corresponding to database operations, clock event detector for detecting clock events, application event detector etc.) and receive event occurrences and associated parameters. In addition, RuMES has to interact with the transaction manager for creating subtransactions in accordance with the coupling specification of rule execution and the object manager indicating what primitive events to be monitored and for executing functions on objects. Finally, RuMES has to interface with the knowledge model/User interface for transforming certain specifications into ones understandable by the system.

Figure 1. illustrates the intercation of RuMES with the other components of a DBMS. The components of RuMES are shown in Figure 2. Briefly, it consists of: a signal manager, a rule manager, and a rule processor composed of an optimizer and a rule evaluator. The task of the signal manager is to receive signals (event_id + descriptive data associated with the event) as they asynchronously arrive from event detectors and signal managers, combine them, and further group them for submission to the rule manager. The signal manager is primarily responsible for detecting events that are recognized

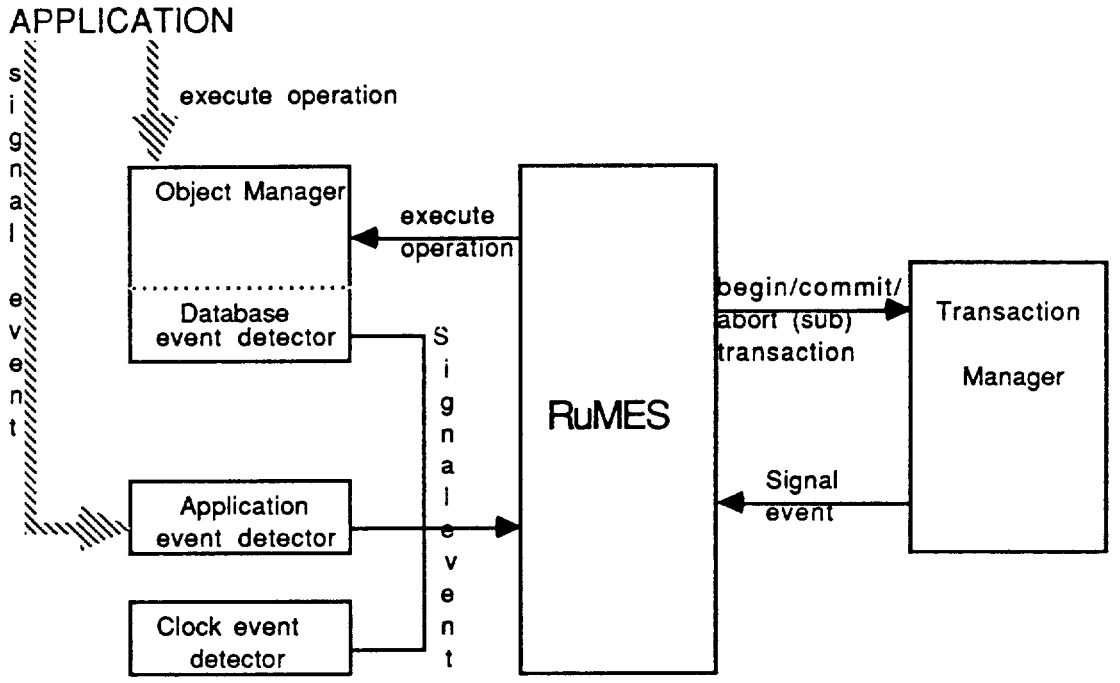


Figure 1. Interaction of RuMES with other DBMS Components

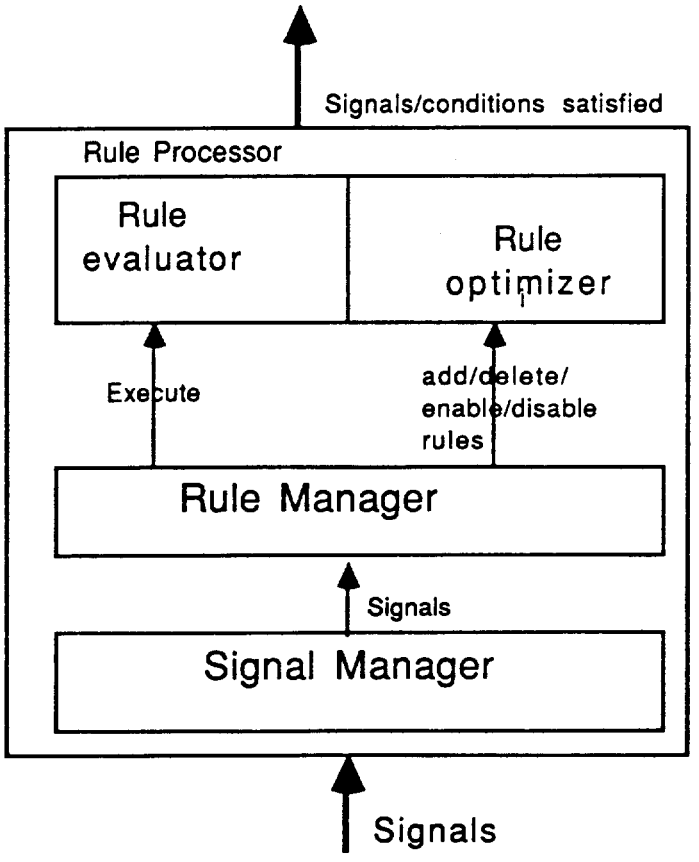


Figure 2. Functional Components of RuMES

by the DBMS and generating associated descriptive data. The rule manager then examines relations to determine which rules need to be evaluated.

The rule manager is responsible for the management of rules in general. This includes initial processing of rules, grouping them for optimization, maintaining the correspondence between rules and their computation graphs, and supporting rule manipulation (add, deletion, enable, and disable).

The rule processor is the analog of a query processor whose task is to apply optimizing transforms to computation graphs. The algorithms used for the efficient evaluation and incremental manipulation of computation graphs are also part of the rule processor.

References

[DARN87] M. Darnovsky, J. Bowman. "TRANSACT-SQL USER'S GUIDE." Document 3231-2.1. Sybase Inc., 1987.

[DAYA88a] Dayal, U., et al., HiPAC :A Research Project in Active, Time-Constrained Database Management. XAIT Interim Report XAIT-88-02. Computer Corporation of America. 1988.

[DAYA88b] U. Dayal. Active Database management Systems. *Proc. of Conf. of Data and Knowledge Bases*. Jerusalem. 1988.

[DAYA88c] U. Dayal, A. Buchmann, D. McCarthy. "Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database Management System." In *Proceedings 2nd International Workshop on Object-Oriented Database Systems*. Bad Muenster am Stein, Ebernburg, West Germany, September 1988.

[DITT86] K. R. Dittrich, A. M. Kotz, J. A. Mulle. "An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases." *SIGMOD Record 15*, No. 3, 1986, pp. 22-36

[ROSE88] A. Rosenthal and U. S. Chakravarthy. Anatomy of a Modular Multiple Query Optimizer. *Proc. of 14th Int'l Conf. on Very Large Databases*. Long Beach, 1988.

[ROSE89] A. Rosenthal, U. S. Chakravarthy, et al., Situation Monitoring in Active Databases. To appear in the Proc. of the 15th International Conference on Very Large Databases. Amsterdam, The Netherlands, Aug 89.

[STON85] M. Stonebraker. "Triggers and Inference In Database Systems." in *On Knowledge Base Management Systems* (Brodie and Mylopoulos,eds.) Springer-Verlag (1986).

[STON87] M. Stonebraker, M. Hanson, S. Potamianos. "A Rule manager for Relational database Systems." Technical Report, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, 1987.

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.

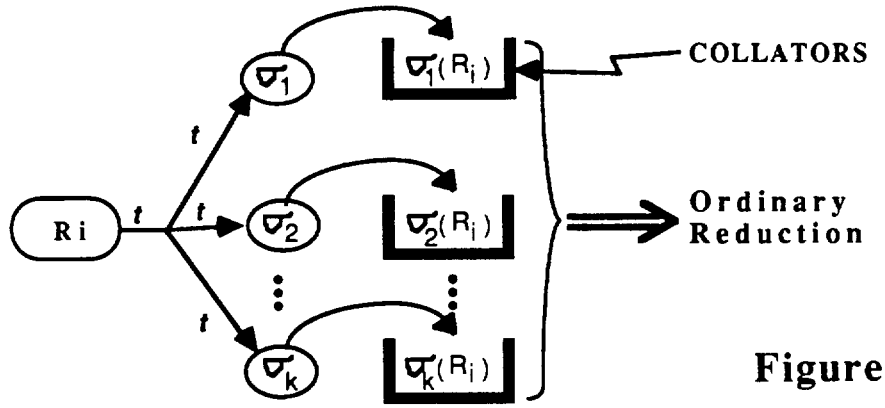


Figure 1. PC-selection

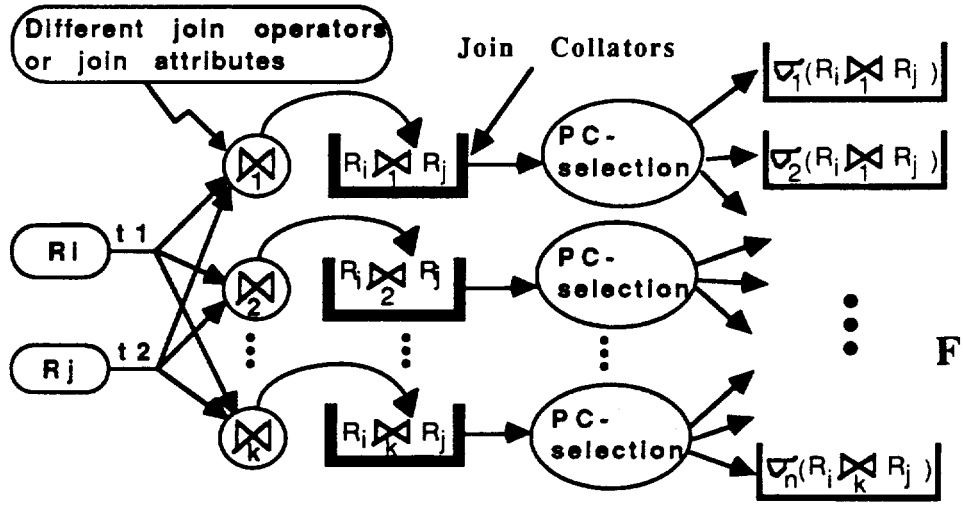


Figure 2. PC-join

Optimization of Nested Tree Queries[1]

M. Muralikrishna

murali@cookie.dec.com
Digital Equipment Corporation
Colorado Springs, CO 80920

Keywords: unnesting, optimization, SQL, the COUNT bug, outer join, anti-join, correlation predicate.

1. Abstract

The SQL language allows users to express queries that have nested subqueries in them. Optimization of nested queries has received considerable attention over the last few years. Most of the previous optimization work has assumed that at most one block is nested within any given block. The solutions presented in the literature for the general case (where an arbitrary number of blocks are nested within a block) have either been incorrect or have dealt with a restricted subset of queries. In this paper we discuss optimization strategies for queries that have an arbitrary number of blocks nested within any given block.

2. Introduction

Traditionally, database systems have executed nested SQL [Astrahan75] queries using Tuple Iteration Semantics (TIS). It was analytically shown in [Kim82] that executing queries by TIS can be very inefficient. It was first pointed out in [Kim82] that nested queries can be evaluated very efficiently using relational algebra operators or set-oriented operations. The process of obtaining set-oriented operations to evaluate nested queries is known as **unnesting**.

It was later pointed out in [Kiessling84] and [Ganski87] that the unnesting techniques presented in [Kim82] do not always yield the correct results for nested queries that have non equi-join correlation predicates or for queries that have the COUNT function between nested blocks. Unnesting solutions for these types of queries were provided in [Ganski87]. These solutions were further refined and extended in [Dayal87].

In this paper, we will focus our attention on unnesting Join-Aggregate (JA) [Kim82] type queries. These queries have correlation join predicates and an aggregate function (AVG, SUM, MIN, MAX, or COUNT) between the nested blocks. The reason for focusing on JA type queries is that many other nesting predicates (such as EXISTS, NOT EXISTS, ALL, ANY) can be reduced to JA type queries [Ganski87, Dayal87]. An example of a JA type query is:

```
SELECT R1.a
FROM R1
WHERE R1.b =
    (SELECT COUNT (R2.b)
    FROM R2
    WHERE R1.c > R2.c
    )
```

The predicate ($R_1.c > R_2.c$) is the correlation join predicate.

We introduce a couple of definitions here:

Definition 1: A **(Nested) Linear Query** is a JA type query in which at most one block is nested within any block.

---

[1]This is the abridged version of the paper that has been accepted for publication at the VLDB conference in August, 1989. The unabridged version also discusses a new dataflow algorithm for the execution of nested tree queries in a multi-processor environment. The new dataflow algorithm cuts down on message and CPU costs over conventional dataflow algorithms.

Definition 2: A **(Nested) Tree Query** is a JA type query in which there is at least one block which has two or more blocks nested within it at the same level.

It is worth pointing out that the unnesting solution presented in [Ganski87] for a linear query with more than two blocks is incorrect (see Section 4). [Dayal87] does not discuss tree queries.

The rest of the paper is organized as follows: In Section 3, we introduce the notation that we will use for JA type queries. In Section 4 we will briefly summarize the results presented in [Dayal87] that enable us to unnest nested linear queries. We will present our solution for tree queries in Section 5.

3. Notation

A JA type query may be represented as a tree. Each node in the tree corresponds to a SQL query block. Query blocks that are nested within a parent block are represented as child nodes of the node corresponding to the parent block. For ease of explanation, we shall assume that each block has one relation in its FROM clause. By definition, a node is also its own ancestor. A predicate clause in a given block may reference a relation associated with any ancestor block. Predicate clauses may either be selection or join predicates.

The relation associated with block (or node) i is represented by R_i ($i > 0$). Lower case letters (a, b, etc.) represent attribute names. A '*' is used to denote all the attributes of a relation. R_i.# is some unique key of R_i. r_i, r_i', r_i'' are each used to denote a tuple of relation R_i. OP_n ($n > 0$) is any one of the following operators ($=, \neq, <, \leq, >, \geq$). $F_i(R_j)$ represents a selection predicate in the ith block on R_j. To simplify the notation, we will assume that all join predicates are binary[2]. A join predicate in the ith block is then represented as $F_i(R_j, R_k)$, where $j, k > 0$ and $j \neq k$. If a predicate in the ith block does not reference R_i, then it is called an outer predicate. In this paper we will assume that there are no outer predicates in our queries. Outer predicates can be handled as shown in [Dayal87].

4. From TIS to Set-Oriented Semantics

In this section, we briefly summarize the general solution presented in [Dayal87] and show how it can be applied to unnest linear queries. For reasons of space constraints, we do not discuss the more specific solutions that are based on the strategies presented in [Kim82]. Besides, the solutions in [Kim82] are not general and hence can be applied only in special cases (as pointed out in Section 2). However, as pointed out in [Dayal87], the unnesting solutions (when applicable) presented in [Kim82] may yield a more efficient execution strategy than the general solution.

Consider the following linear JA type Query.

Example 1: A Two Block Linear Query

```
SELECT R1.a
FROM R1
WHERE F1(R1)
AND R1.b OP1
    (SELECT COUNT (R2.*)
    FROM R2
    WHERE F2(R2) AND F2(R2, R1)
    )
```

An unnesting algorithm would outer join[3] R_1 and R_2 using predicate $F_2(R_2, R_1)$ (after performing the respective selections first). The algorithm would then group the result by R_1.# (some unique key of R_1[4]) and compute COUNT(R_2.*) for each group and select only those groups associated with each tuple of R_1 that satisfy (R_1.b OP_1 COUNT(R_2.*)). Note that an outer join (OJ) is performed to avoid the COUNT bug [Ganski87].

---

[2] n-ary join predicates can be easily incorporated into the solutions presented in this paper.

[3] When we talk about outer joins, we implicitly mean left outer joins.

[4] A unique key is required in order to avoid the problem with duplicates in R_1 [Ganski87].

A linear query with multiple blocks will give rise to a 'linear J/OJ expression' where each instance of an operator is either a join or an outer join. A general linear J/OJ expression would look like:

$$R_1 \text{ J/OJ } R_2 \text{ J/OJ } R_3 \text{ J/OJ } \ldots \text{ J/OJ } R_n$$

Relation R_1 is associated with the outermost block, relation R_2 with the next inner block and so on. An outer join is required if there is a COUNT between the respective blocks. In all other cases (AVG, MAX, MIN, SUM), we need perform only a join. The joins and outer joins are evaluated using the appropriate predicates. Since joins and outer joins do not commute with each other[5], a legal order may be obtained by computing all the joins first and then computing the outer joins in a left to right order (top to bottom if you like) [Dayal87]. Thus, the expression R_1 OJ R_2 J R_3 J R_4 OJ R_5 J R_6 can be legally evaluated as ((R_1 OJ (R_2 J R_3 J R_4)) OJ (R_5 J R_6)). Since we can evaluate joins in any order, we can choose the cheapest join order to join R_2, R_3, and R_4.

It is worth pointing out here that the solution presented in Section 9 of [Ganski87] for multiple level queries was incomplete in the sense that it does not discuss legal orderings when joins and outer joins are present in the same expression.

After all the joins and outer joins have been evaluated, the aggregate functions are evaluated in a bottom-up order after grouping the result by the appropriate unique keys. We will illustrate these ideas using the query of Example 2.

Example 2: A Three Block Linear Query

```
SELECT R1.a
FROM R1
WHERE F1(R1)
AND R1.b OP1
    (SELECT COUNT (R2.*)
    FROM R2
    WHERE F2(R2) AND F2(R2, R1)
    AND R2.c OP2
        (SELECT (COUNT(R3.*)
        FROM R3
        WHERE F3(R3) AND F3(R3, R2) AND F3(R3, R1)
        )
    )
```

The corresponding linear expression is R_1 OJ R_2 OJ R_3 and hence a legal order is (R_1 OJ R_2) OJ R_3. The predicate for R_1 OJ R_2 is $F_2(R_2, R_1)$ and the predicate for the outer join with R_3 is $F_3(R_3, R_2)$ AND $F_3(R_3, R_1)$.

We now show how the query of Example 2 can be evaluated using set-oriented operations. The result is obtained by executing more than one query. The result from one query may be pipelined to the next query. The two queries in this case are (not in strict SQL syntax!):

Query A:

```
SELECT INTO TEMP
R1.#, R1.a, R1.b, R2.*
FROM R1, R2, R3
WHERE (R1 OJ R2) OJ R3
GROUP BY R1.#, R2.#
HAVING R2.c OP2 COUNT(R3.*)
```

[5]Dayal proposed the notion of generalized joins (G-Joins) to make joins and outer joins commutable but the equation given in the paper was incorrect. Without repeating the notation used in defining G-Join and the formal definition of G-Join, we simply state that the following equation was given in [Dayal87]: G-Join (R, G-Join(S, T; ∅; J2); R.*; J1) = G-Join(G-Join(R, S; R.*; J1), T; R.*; J2). However, it can be shown that this equation does not hold for the query in Figure 4.1 on page 202 in Dayal's paper.

Query B:

SELECT R_1.a
FROM TEMP
GROUP BY R_1.#
HAVING R_1.b OP_1 COUNT(R_2.*)

The results from Query A are fed into Query B. Even though the selection predicates ($F_i(R_i)$, i = 1, 2, 3) have not been shown in Query A, they are applied to the respective relations before they participate in the outer joins. The outer join predicates are also implicit in Query A.

4.1. A Few Subtleties

Query A has a few subtleties that were not mentioned in [Dayal87] and deserve to be highlighted. These subtleties will lead us to the development of the new dataflow algorithm (discussed in the VLDB paper). The outer join between R_1 and R_2 results in two sets of tuples, viz., (R_1- X NULL)[6] and R_1R_2. R_1R_2 denotes the set $\{(r_1, r_2)$: $F_2(R_2)$ AND $F_2(R_2, R_1)$ AND $F_1(R_1)\}$, where the r_1 tuple $\in R_1$ and the r_2 tuple $\in R_2$. Let R_1+ denote the set of tuples of R_1 present in R_1R_2 (tuples of R_1 that participated in the join with R_2). R_1- denotes the set R_1 - (R_1+) (the tuples of R_1 in the anti-join).

Similarly, let $R_1R_2R_3$ denote the set $\{(r_1, r_2, r_3)$: $F_3(R_3)$ AND $F_3(R_3, R_2)$ AND $F_3(R_3, R_1)$ AND $F_2(R_2)$ AND $F_2(R_2, R_1)$ AND $F_1(R_1)\}$. Let the set of (r_1, r_2) tuples in R_1R_2 that joined with at least one tuple of R_3 be denoted by R_1R_2+. The set of (r_1, r_2) tuples that did not join with any tuple of R_3 is denoted by R_1R_2- and is equal to R_1R_2 - (R_1R_2+). Thus, the outer join with R_3 may yield up to three distinct sets of tuples, viz., (R_1- X NULL X NULL), (R_1R_2- X NULL), and $R_1R_2R_3$ respectively.

The (GROUP BY ... HAVING) operation in Query A has special semantics associated with it. For a given group of (r_1.#, r_2.#), if (r_2.c OP_2 COUNT(R_3.*)) is true, the (r_1.#, r_2.#) group is passed along to Query B. However, if (r_2.c OP_2 COUNT(R_3.*)) is false, the (r_1.#, r_2.#) group cannot be discarded. If the (r_1.#, r_2.#) is discarded and if this is the only group in which r_1 was present, COUNT(R_2.*) associated with the r_1 tuple is 0 and hence should be preserved. If (r_1.b OP_1 0) is true, r_1 will be part of the result. The (r_1, r_2) tuple that does not satisfy (r_2.c OP_2 COUNT(R_3.*)) should be passed along to Query B as (r_1, NULL). Similarly, for tuples in the set (R_1- X NULL X NULL), the GROUP BY ... HAVING operation passes them as (R_1- X NULL) to Query B because the predicate (r_2.c OP_2 COUNT(R_3.*)) is false as (NULL OP 0) is false.

5. Tree Expressions or Non Linear Expressions

So far we have restricted our discussion to linear queries only. If we permit more than one block to be nested within a given block at the same level (nested tree queries), we can get J/OJ expressions that are arbitrary trees. In this section, we will extend Dayal's solution to tree queries. A simple extension, albeit inefficient, to evaluate a tree expression would be the following: Choose an arbitrary path (perhaps the least expensive one) from the root to a leaf and evaluate the linear expression specified by this path as outlined in the previous section. This will yield a subset R_1' of the tuples of the root relation R_1. Using tuples in R_1', another path is evaluated yielding R_1'', a subset of R_1'. This is repeated until all paths are exhausted and the final set of result tuples are obtained.

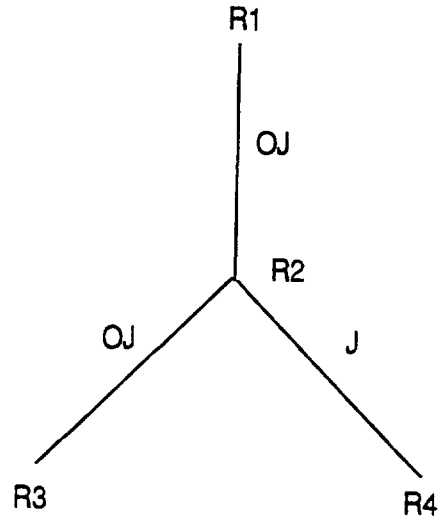
The above scheme is inefficient because relations that belong to two or more paths will be accessed more than once. For example, consider the tree query shown in Example 3 whose tree expression is shown in Figure 1. The edges in Figure 1 are labeled either by a J (denoting a join) or by an OJ (denoting an outer join).

[6] X represents the cartesian product operation.

Example 3: A Tree Query

```
SELECT R1.a
FROM R1
WHERE F1(R1)
AND R1.b OP1
    (SELECT COUNT (R2.*)
    FROM R2
    WHERE F2(R2) AND F2(R2, R1)
    AND R2.c OP2
        (SELECT (COUNT(R3.*)
        FROM R3
        WHERE F3(R3) AND F3(R3, R2) AND F3(R3, R1)
        )
    AND R2.d OP3
        (SELECT (AVG(R4.d)
        FROM R4
        WHERE F4(R4) AND F4(R4, R2) AND F4(R4, R1)
        )
    )
```

Figure 1: Tree Expression for the Query in Example 3.



Assume that the first path chosen is R_1--R_2--R_3. After evaluating the expression (R_1 OJ R_2) OJ R_3 and computing the respective aggregates bottom up, we will get a subset R_1' of tuples. Using these tuples, we take the other path. The J/OJ expression along this path is R_1' OJ (R_2 J R_4). Thus, the relation R_2 is accessed again. It would be ideal if each relation is accessed only once. After evaluating a J/OJ expression along one path, we would like to compute the aggregates bottom up only up to the point where a new branch begins. The idea is to use the tuples obtained thus far to evaluate the remainder of the J/OJ expression along the new path. However, this cannot be accomplished in a straight forward manner. Two kinds of anomalies may occur. We will illustrate these using the tree expression of Figure 1.

After evaluating (R_1 OJ R_2) OJ R_3 and the aggregate COUNT(R_3.*), there are two distinct sets of tuples. Tuples in the first set are of the form (r_1, NULL) and tuples in the second set are of the form (r_1, r_2) where $r_1 \in R_1$ and $r_2 \in R_2$.

Anomaly 1: Consider a tuple (r_1', NULL) from the first set. If r_1' is not present in the second set, COUNT(R_2.*) associated with r_1' is 0 and will be part of the result if (r_1'.b OP_1 0) is true. However, if (r_1', NULL) is joined with R_4 when the second path is taken, r_1' will be lost (because $F_4(R_4, R_2)$ is false since the R_2 fields are NULL).

Anomaly 2: Consider a tuple (r_1'', r_2') from the second set. If there is no tuple of R_4 such that $F_4(R_4, R_1)$ is true, then the r_1'' tuple will be lost after the join is computed. However, if the second path (R_1--R_2--R_4) was taken first, we would have got (r_1'', NULL, NULL) after evaluating R_1 OJ (R_2 J R_4). In this case, we do not lose r_1''. When the other path (R_1--R_2--R_3) is taken, r_1'' will not be lost as there is already an outer join between the R_2 and R_3 blocks.

The two anomalies demonstrate that tuples will be lost if a join is performed after evaluating an outer join. These tuples can be saved if the join between the R_2 and R_4 blocks is performed as an outer join. This leads to the following lemma.

Lemma: Let R_1--R_2--...--R_i--...--R_p and R_1--R_2--...--R_i-J-R_j--...--R_q be two paths from the root R_1 to the leaves R_p and R_q respectively in a tree expression. Let there be at least one outer join in the shared path R_1--R_2--...--R_i. Assuming we chose the R_1--R_2--...--R_p path first and there is a join between R_i and R_j, we can obtain the correct result by treating the join between R_i and R_j as an outer join.

Note that any joins below R_j are not affected as they will be evaluated before the outer join between the R_i and R_j blocks.

The proof for the lemma can be obtained by following the train of thought of the previous paragraphs and is omitted here. In summary, any join that comes after an outer join in a path must be

evaluated as an outer join. Then each relation need be accessed only once.

6. Acknowledgments

The author wishes to thank Bob Gerber, Goetz Graefe, Krishna Kulkarni, Shirish Puranik, Jim Reuter, Mateen Siddiqui, Lynn Still, and Wai-Sze Tam for carefully reviewing earlier versions of this paper. Their efforts have considerably improved the quality of the paper.

7. References

[**Astrahan75**] M. Astrahan, and D. Chamberlin, "Implementation of a structured English query language," Comm. of the ACM, Vol. 18, No. 10, (October 1975).

[**Dayal87**] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers," Proc. VLDB Conf., pp.197-202, (September 1987).

[**Ganski87**] Richard A. Ganski and Harry K. T. Long, "Optimization of nested SQL Queries Revisited", Proc. SIGMOD Conf., pp. 23-33, (May 1987).

[**Kiessling84**] W. Kiessling, "SQL-like and Quel-like correlation queries with aggregates revisited", UCB/ERL Memo 84/75, Univ. of California at Berkeley, (Sept. 1984).

[**Kim82**] W. Kim, "On Optimizing an SQL-like Nested Query", Trans. on Database Systems, Vol 9, No. 3, (1982).

BLAST Query Optimization

Bradley Hammond
ShareBase from Britton Lee, Inc.

1. Overview of the BLAST Optimizer

The goal of the BLAST project is to build an ANSI SQL database system, including the referential integrity addendum. The project includes a new query optimizer, parser, semantic checker, and executor. It is a successor to an IDL database system, and many of the access routines have been kept.

The optimizer works with the following data structures:

Query Tree: A representation of the query whose structure closely corresponds to the BNF structure of SQL. This is the input to the optimizer.

Query Graph: The query is represented as a partially ordered list of relational operations. The partial ordering of the operations corresponds to the data flow constraints imposed by the query, i.e. the partial order guarantees that a value is not used before it is computed. While we have made some refinements to it, the query graph representation is based on [DAYA87].

Plan Tree: The specification of how the query is to be processed. This is the output of the optimizer. Each node of the plan tree specifies the implementation details for an operation, and the tree hierarchy specifies the order of operations.

The optimization process consists of the following four components

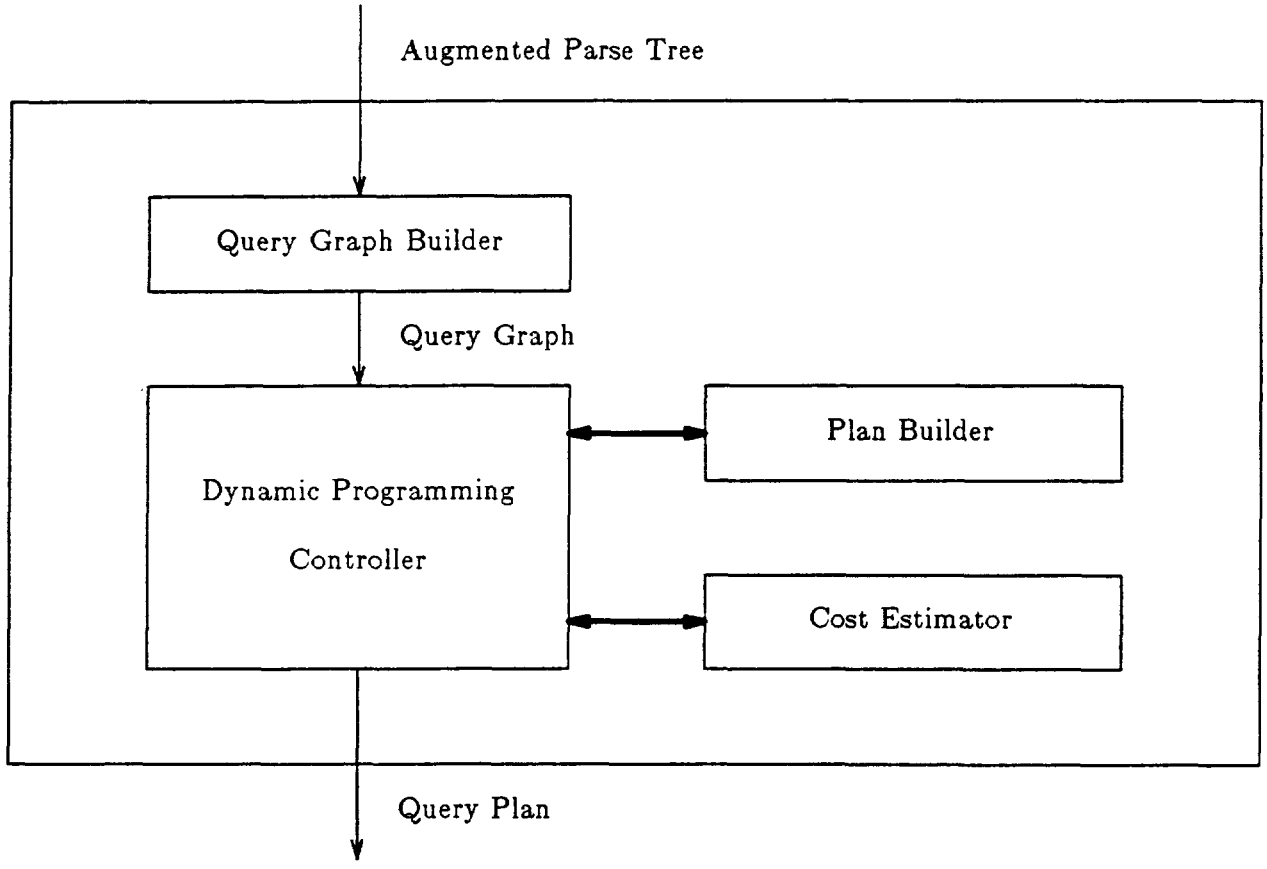
Query Graph Builder: A query tree is converted into a query graph.

Plan Building: Given a partial plan and a new table to add, the plan builder generates a number of new plans. For example, to join a new table onto a partial result, it generates a plan that uses a nested loop join and a plan that uses a merge join.

Cost Estimator: The Cost Estimator estimates the costs of executing a plan.

Dynamic Programming Control: The optimizer uses dynamic programming techniques to search the space of all possible join orders. It does a breadth first search and incrementally builds the best plan for the query. At each level of the search it calls the plan builder and cost estimator in order to generate the best sub-plan.

The following diagram shows the basic structure of the BLAST Optimizer.



2. The Optimizer's Input: Query Graphs

The BLAST optimizer uses a query graph structure that has two basic elements: Restrictions and Nodes. In turn, a node may represent either a base table or an aggregate formed from one or more base tables. While [DAYA87] uses edges to represent

joins, and directed edges to represent outer joins, this imagery starts to wear a bit thin with complex restrictions such A.x = B.x + C.x.

In cases where restrictions tie together more than two nodes, it is difficult to illustrate the relationship with nodes and edges, but the internal representation of such complex restrictions is fairly simple. The nodes that participate in a restriction are represented by a precedence bitmap. Bitmaps that represent precedence are also embedded in nodes, for example when certain outer restrictions must be applied before computing an aggregate.

As another consequence of the fact that restrictions can connect more than two nodes, the restrictions don't carry the notion of semi-join or outer-join. The relation of tables with respect to semi-ness and other qualities is represented by sets of level numbers in the nodes of the graph.

For example, take the following query:

```
SELECT * FROM A, B WHERE
        EXISTS (SELECT * FROM C WHERE C.y = A.y + B.y);
```

The BLAST optimizer represents this query with a query graph containing three nodes and one restriction. The semi-ness of the relation between the tables is represented by giving tables A and B a semi-level of 0 and table C a semi-level of 1. One plan that would be considered would be to join tables A and B (Cartesian product) and then doing a semi-join between the result and table C.

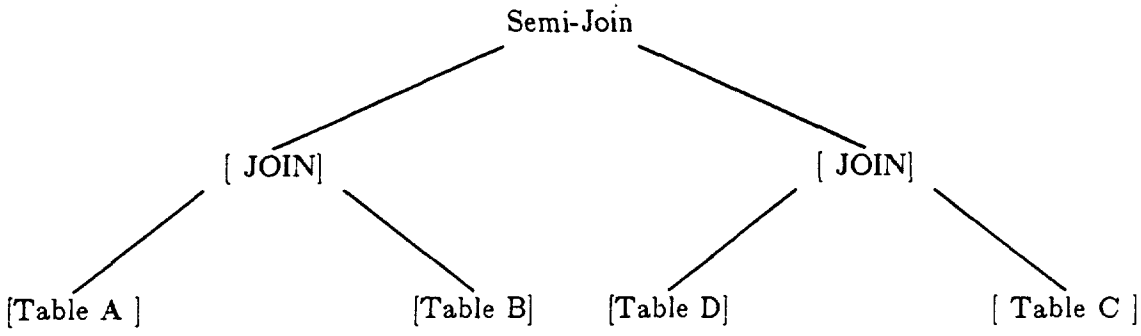
3. The Optimizer's Output: Plantrees

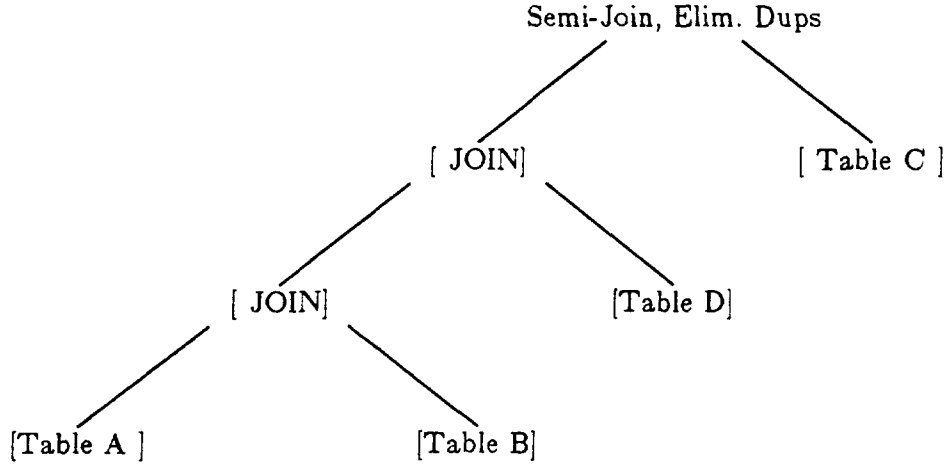
The BLAST optimizer produces plan trees containing nodes that represent such basic objects as index scans, heap scans, nested loop joins, merge joins, unions, "exists" and "not exists" nodes, sorts, and aggregates. Generally plan tree nodes have two or fewer children, although unions are sort of an exception. (A union of n objects is originally built as a tree of n binary union nodes, which are subsequently collapsed into a single object with n children.) Joins may have modes, such as semi-joins and outer-joins, which is independent of the method contained in the join node. The "exists" and "not exists" nodes are used extensively in the checking of referential integrity constraints during deletes, inserts, and updates.

3.1. Shape Of Plan Trees

One interesting feature of the plan trees produced by the BLAST optmizer is that the shape of the trees may not always be linear. This is different from many systems, see for example [ORAC86] and [PONG88]. For example, consider the following query and plantrees:

```
SELECT * FROM A, B WHERE
        EXISTS (SELECT * FROM C, D
                WHERE ...);
```





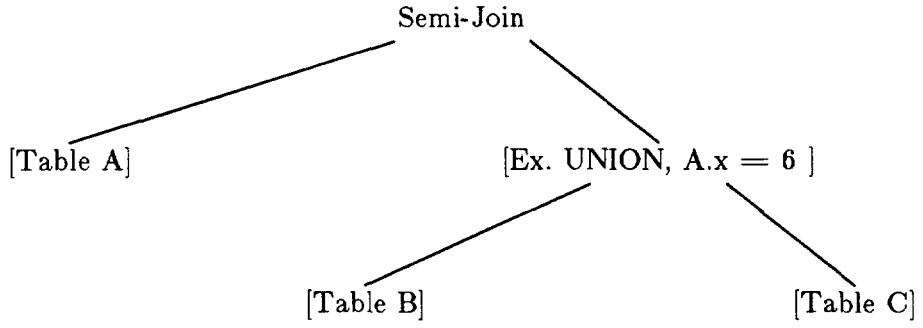
While there are other valid plans that will be considered, the first tree is always more efficient than the second. The advantages of the non-linear tree lie in the semantics of the semi-join. The BLAST executor uses tuple substitution, so that values of tables A and B can be used in the scans over D and C. The semi-join at the top of the tree ensures that once a given pair of tuples from A and B have been qualified, the executor will only search for a single matching C,D tuple pair. If the executor had to work from the linear version of the plan, then the scan over table D could not stop until all matching tuples were found. The generation of duplicates and subsequent elimination of duplicates is less efficient than using the non-linear tree. In similar ways, the shape of the tree can preserve the semantics of user specified outer-joins, while tuple substitution allows efficient use of indices in scanning tables.

3.2. Use of Existential UNIONS

One of the useful plan tree nodes is an existential UNION. The existential UNION node differs from a usual UNION node in that its subscans need not be union-compatible, no data is passed up, and they are only used to check whether all of the sub-scans are empty. To see how these existential UNIONS are useful in queries, consider the following query:

```
SELECT * FROM A WHERE (A.x = 6) OR
        EXISTS (SELECT * FROM B WHERE ...) OR
        EXISTS (SELECT * FROM C WHERE ...);
```

If the first clause were not present, we could simply do a semi-join between A and the existential UNION of the scans on B and C. To optimally execute the full query, we should only scan B or C when (A.x = 6) is false. Accordingly, we have extended the existential UNION to include a predicate. Thus the plan for this query becomes:



4. The Optimizer Framework

The BLAST query optimizer uses a dynamic programming approach to choose optimal plans. Plans are built in a tree structure, and during intermediate stages of optimization several trees can share children. For example, if the optimal plan for computing (A join B) has been determined, then this plan might be a subplan for both (A join B join C) and (A join B join D). The optimizer maintains two types of structures that point to plans, a hash table and priority queues. The hash table facilitates looking for the optimal plans corresponding to various combinations of tables that have already been examined. The priority queues are used for pruning the set of plans which will be pushed forward and developed further.

The combination of a pruning technique in conjunction with dynamic programming implies that under some circumstances the pruning might prevent finding the optimal

plan. However, for a sufficiently large query, any completely exhaustive search will require either too much memory or too much time to be practical in a high performance database machine. The parameters of pruning can tuned, and for reasonably small queries the search can be exhaustive.

The dynamic programming framework is based on incrementally adding tables, and is not constrained to follow join edges. One perhaps surprising advantage of this method is that Cartesian products are automatically considered. The following example, inspired by queries from customers' applications, shows a case in which Cartesian products are part of the optimal plan:

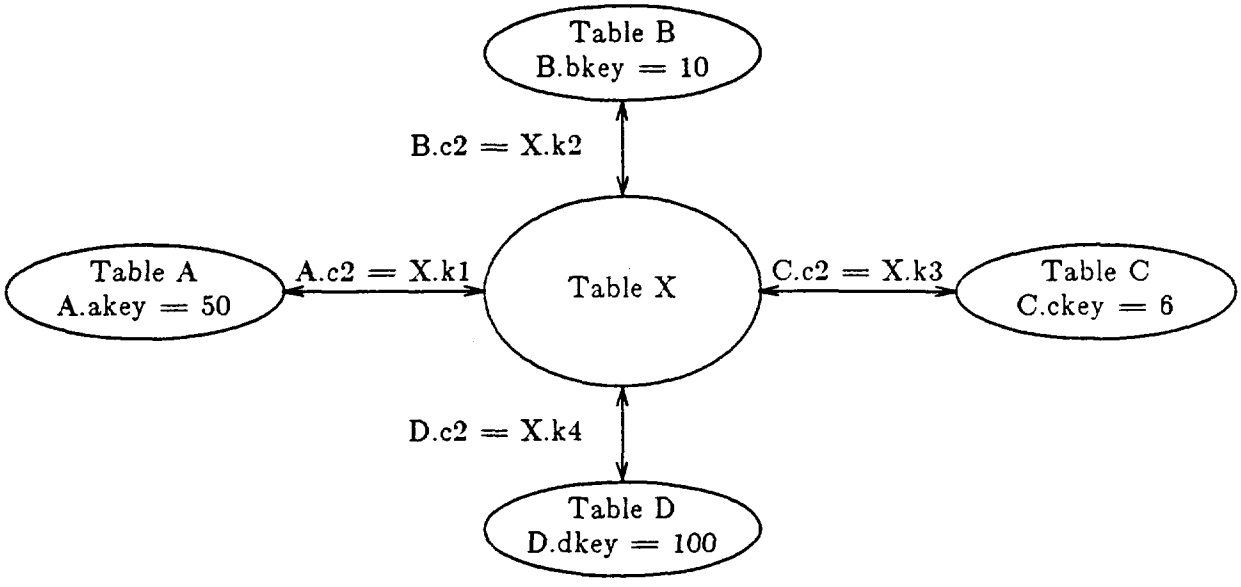


Table X in the above diagram is very large, and has an index consisting of the four columns k1, k2, k3, and k4. The other tables will return very few tuples after the applicable restrictions are applied. The best plan is to access table X last, and use the complete key. It would be simple for a rule based system to determine that it is better to access Table A before accessing Table X, but it might require a lot of special cases to force consideration of performing the Cartesian products between A, B, C and D before accessing Table X. The BLAST optimizer automatically pushes forward all of the valid plans, and needs no sophisticated rules to determine when Cartesian products are part of a good plan. In cases where Cartesian products are not desirable, the BLAST optimizer drops them when the strategies become dominated by other plans.

5. References

[DAYA87] Dayal, U. "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers" Proc. 13th VLDB conference 1987.

[ORAC86] "Oracle Database Administrator's Guide". 1986-87.

[PONG88] Pong, M. "NonStop SQL Optimizer: Query Optimization and user Influence" Tandem Systems Review 1988.

Optimizing SQL Queries for Parallel Execution

Günter von Bültzingsloewen

Forschungszentrum Informatik an der Universität Karlsruhe
Haid-und-Neu-Straße 10–14, D–7500 Karlsruhe 1

Abstract: *The optimization problem discussed in this paper is the translation of an SQL query into an efficient parallel execution plan for a multiprocessor database machine under the performance goal of reduced response times as well as increased throughput in a multiuser environment. We describe and justify the most important research problems which have to be solved to achieve this task, and we explain our approach to solve these problems.*

1 Introduction

Recently, several experimental database machines (e.g. ARBRE [Lori89], BUBBA [AlCo88, CABK88], GAMMA [DeWi86] and KARDAMOM [Bült89]) have been designed which use parallelism to obtain increased performance (both response times and throughput). They are based on a loosely or closely coupled multiprocessor system. A disk can be accessed only by the processor it is attached to (i.e. no disk sharing). Disks may be attached to some or all of the processors. Relations are partitioned and spread over several disks.

Parallelism is exploited at two levels (at least): we have inter-transaction parellelism for increased throughput as well as intra-query (and thus intra-transaction) parallelism on the level of set-oriented algebraic operations for reduced response times. A query execution plan in such a system can be represented by a data flow program, a directed acyclic graph whose nodes represent operations and whose arcs represent the flow of data between operations [Chan76, BoDe82]. Independent operations of a dataflow program can be executed in parallel. Additional potential parallelism can be obtained in the following ways:

- *Change order of operations:* An example of this kind of transformation is the use of flatter and bushier join trees compared to the linear trees often used in today's optimizers (e.g. [Seli79]).
- *Node splitting:* Operations, which access or manipulate base relations, have to be split into suboperations according to data distribution: each partition of a base relation resides on a disk attached to a certain processor and can be accessed only by this processor. Additional node splitting can be performed for operations like join and aggregation which operate on intermediate relations produced by former operations (e.g. parallel join algorithms [DeGe85, KTMo83, RiLM87]).
- *Pipelining:* The (pseudo-)parallel execution of operations which follow each other is already useful in a single processor execution plan: main memory requirements for intermediate results are reduced, hence the swapping of intermediate results to disk may be avoided. In a parallel execution plan, besides allowing additional parallelism, it is a choice to optimize communication: size of messages vs. number of messages. Therefore, it is always advantageous to use pipelining if intermediate results are large enough, and the possibility of pipelining should always be marked in a query execution plan.

In order to express how much of the potential parallelism is actually realized, the dataflow program can be augmented by scheduling information (e.g. assignment of operations to processors, priority of operations).

The optimization problem we are concerned with is: Determine an efficient parallel execution plan for a given query using the above possibilities, which can afterwards be executed several times (i.e. precompilation of queries). In the remainder of this paper, we describe and justify the most important research problems which have to be solved to achieve this task, and we explain our approach to solve these problems.

2 Objective of Optimization

Usual objectives of query optimization are to minimize total processing cost (a weighted sum of CPU-cost, I/O-cost, and communication cost) or to minimize execution time. In centralized database systems, these objectives are largely equivalent, and optimizers attempt to minimize total processing cost [JaKo84]. In a multiprocessor database machine, these objectives do not coincide as a higher degree of parallelism usually implies reduced execution times and increased total processing cost because of increased processing, communication and control overhead. Hence we have

Problem 1: *How should the objective of optimization in a multiprocessor database machine be defined?*

Proposed Solution: The overall performance goal of a multiprocessor database machine is to obtain increased throughput as well as reduced response times in a multiuser environment. Throughput is reduced if we increase the parallelism inside single queries because of increased processing overhead; average response time is usually reduced because of reduced execution time. However, in case of high resource utilization (high throughput) it may actually be increased as higher total processing cost may cause longer waiting times until a query can be executed. Hence, the optimization problem involves a tradeoff between execution time and total processing cost. Which combination leads to the desired throughput and response times depends on the load conditions and can only be determined at run time. Therefore, the optimizer should construct several alternative plans under the following objectives:

(1) Minimize total processing cost.

(2) Minimize execution time subject to a maximum x%-increase of total processing cost (for several values of x); or equivalently

(2') Maximize the degree of parallelism (defined as the average number of processors used during the execution of the query, i.e. total processing cost divided by execution time) subject to a maximum x%-increase of total processing cost.

To choose an appropriate plan at runtime, we have to know more about the effects of increased total processing cost and reduced execution time on response time and throughput. In order to obtain this knowledge, simulation experiments or performance measurements should be performed, which examine query execution in a multiprocessor database machine under several load conditions using the query execution plans determined by the optimizer.

3 Architecture of the Optimizer

The generation of an optimal query execution plan is a complex (NP-hard) optimization problem. Solution procedures for such problems have been studied extensively in Operations Research and Artificial Inelligence [Pear84]. They can be described using the following basic elements:

- A *code* which can represent each *object of the search space* (solution candidate, set of possible solution candidates).
- A *rule base* containing rules to transform the encoding of one object to that of another object in order to scan the search space.
- A *search strategy*, i.e. an effective method to select the next transformation rule to be applied to an object.

- *Cost functions* which serve to evaluate objects of the search space.

We can distinguish two basic possibilities to organize the search space:

(1) *Stepwise Improvement:* Objects represent complete solution candidates. Starting from an initial solution, transformations are applied in order to obtain improved solutions.

(2) *Split-and-prune:* Objects correspond to subsets which contain all potential solutions which are derivable from a certain partial solution. Transformations split subsets into smaller subsets (extend partial solutions). Subsets which presumably do not contain an optimal solution can be pruned, i.e. excluded from further expansion. If only such subsets which certainly do not contain an optimal solution are pruned, we obtain a *branch-and-bound* algorithm.

Both possibilities have been used in query optimization. Examples of the use of stepwise improvement are [IoWo87, SwGu88, GrDe87], examples of branch-and-bound (or the corresponding dynamic programming algorithm) are [Shan88, LeFL88]. Several search strategies can be used in combination with these possibilities: unsystematic search (e.g. local search, simulated annealing), blind, systematic search (depth-first search, breadth-first search) and informed, systematic search (best-first search, A*-algorithm). Up to now, there is no clear evidence which choice is best. Hence we have

Problem 2: *Should a query optimizer use stepwise improvement or split-and-prune (branch-and-bound, dynamic programming)? If either possibility is chosen, which search strategy should be used?*

Proposed solution: We choose to investigate split-and-prune strategies for one main reason: it leads to a better analysis of the optimization problem, as we have to define precisely the decisions involved in the generation of a parallel execution plan. Thus we can possibly exercise more control on the solution candidates examined during the search; to accomplish this, we have to develop heuristics to prune the search space and to evaluate objects in order to perform a best-first search.

We can organize the search as a number of steps, each of which makes certain decisions and thus reduces the number of plans obtainable from a partial solution:

(1) Generate all reasonable algebraic expressions for a given SQL query. An algebraic expression corresponds to the set of all query execution plans, which perform the algebraic operations in the same order as in the expression.

(2) Generate execution methods for the operations of an algebraic expression. The result of this step is a single processor execution plan.

(3) Generate parallel execution plans using node splitting and determining scheduling information. According to the different objectives, several execution plans are generated.

Code, rule base and search strategy used in these steps are discussed in the following sections.

4 Generating Single Processor Execution Plans

In conventional implementations of SQL, queries are executed at least partially tupel-oriented, i.e. by nested iteration (System R, System R*). In contrast to that, we want to generate execution plans that contain only (methods for) set-oriented algebraic operations. All known approaches to translate SQL queries into relational algebra [CeGo85, LeVi85, Daya87, Kim82] are incomplete and some are partially incorrect (see [Bült87]). Furthermore, all approaches except [Daya87] generate very complicated expressions which are no good starting point for further optimization. Hence we have

Problem 3: *How can we generate all reasonable algebraic expression for a given SQL query in a systematic way?*

Proposed solution: The generation of algebraic expressions is performed in three steps:

(1) An SQL query is translated into an expression of an extended relational algebra (extension of [Klug82]). The expression is in a special normal form which defines an order of algebraic operations only as far as absolutely necessary, so that an optimal expression can be reached from this starting point. The translation has been developed starting from [Bült87], however avoiding unnecessarily complicated expressions.

(2) The expression is simplified using transformations which are known to produce better expressions.

(3) Starting from this normal from, algebraic expressions and methods for the execution of algebraic operations are generated using functional rules similar to [LeFL88].

The normal form we use is a union of expressions

$$\pi_{X(Y)}\sigma_{\psi_1}(\phi_{<X_g,F>}\sigma_{\psi_2}((R_1 \times R_2 \times \ldots R_n) \mid <S_1[e_1],\ldots,S_m[e_m]>) \mid <S'_1[e'_1],\ldots,S'_l[e'_l]>).$$

Projections are extended to retain duplicates ($\pi_{X(Y)}$ denotes a projection on X which retains duplicates according to the number of different Y-values). An extended selection predicate ψ is a conjunction of simple predicates (without logical connectives). However, in case of a selection on a single relation ($n = 1$), it may also contain disjunctions to avoid unnecessary unions. An extended selection predicate may contain join and semijoin conditions. The latter are expressed as $[\exists S_i : \psi_i]$ or $[\neg\exists S_i : \psi_i]$, indicating a positive or negative semijoin of $(R_1 \times R_2 \times \ldots R_n)$ with e_i. The motivation behind this unfamiliar notation is that we wish to perform several semijoin operations on a relation in a single filter operation.

$\phi_{<X_g,F>}(e)$ denotes the aggregation, which groups e by X_g-values and applies the aggregate functions F to each group. In semijoin expressions e_i, we also use a generalized aggregation $\Phi_{<X_g,F>}(e_1 \rhd e_2)$ which attaches aggregate function values to each tuple t of e_1 by applying F to the group of tuples t' of e_2 having identical X_g-values ($t[X_g] \equiv t'[X_g]$). It is used in the translation of subqueries with aggregate functions and is equivalent to an outer join followed by the standard aggregation $\phi_{<X_g,F>}$. Arithmetic expressions may be used in projections, selection predicates and aggregations.

Transformations performed in the second step include the replacement of a generalized aggregate formation by an aggregation $\phi_{<X,F>}$ (without using an outer join); the elimination of unneccesary join predicates; and the replacement of two negative semijoins by a division. All of these transformations are only applicable under certain restricted conditions.

The generation of algebraic expressions starting from this normal form requires the following decisions: In conjunctive expressions, the main decision is the ordering of join operations. Furthermore, we have to decide wether selections are performed before or after a join. Aggregation operations have to be positioned with respect to join operations. We may use the generalized aggregation in combination with simple joins or the standard aggregation in combination with outer joins.

Algebraic expressions whose results are combined by a union may be optimized independently, whereupon subexpressions common to several algebraic expressions can be combined. An alternative to this appraoch is the optimization of the complete expression in one step. The advantage of this alternative is that common subexpressions can be produced intentional, possibly yielding a better expression. However, it is an open problem how to accomplish this.

5 Generating Parallel Execution Plans

To generate parallel execution plans we can introduce additional potential parallelism by changing the order of operations and by node splitting (assuming the possibility of pipelining is already marked in the single processor execution plan). A change in the order of operations is considered automatically, if we parallelize not only the most cost effective single processor execution plan, but also the second, third, etc (but only such plans, where the order of operations and not only the execution methods differ). Hence the only change in the underlying dataflow program is node splitting which may be enforced (access of base relations) or optionally chosen (other operations).

In order to evaluate total processing cost of a dataflow program, we have to assign the operations to processors (assuming that an operation is executed completely by one processor), as the cost for local communication on one processor is different from the cost for remote communication. The minimization of total processing cost is thus similar to query optimization in distributed database systems (however, communication is usually not the most important cost factor).

In order to evaluate execution time, we have to build a schedule as in deterministic scheduling theory [Coff76]. Besides the assignment of operations to processors, it also assigns execution intervals to each operation. Minimization of execution time is thus similar to the problems studied in scheduling theory; it differs mainly in the possibility of pipelining and node splitting, and in a more complicated cost model including CPU-cost, I/O-cost and communication.

The scheduling decisions underlying the evaluation of execution time have to be transformed into scheduling information, upon which runtime scheduling can be based. As cost estimations are generally imprecise, it does not make sense to fix execution intervals for each operation. Instead, we need a more abstract description of a parallel execution plan, from which execution intervals can be deduced for cost evaluation and which can be used for runtime scheduling.

Problem 4: *What scheduling information should parallel execution plans contain?*

Proposed solution: We assume that an operation is executed by a process running on a certain processor and can be interrupted only, if it is waiting for a certain event (I/O, operand, pipelining), or if an operation (process) with a higher priority becomes executable (usually fulfilled by a real time operating system). Under these assumptions, runtime scheduling decisions are based upon the following information:

(1) *Assignment of operations to processors.* It has to be fixed only for operations that access base relations. All other nodes can be collected into groups which are mapped to processors at runtime, at most one group to one processor. Thus a limited dynamic load balancing at runtime is possible besides the choice of an appropriate execution plan.

(2) *Priority of an operation.* It is assigned to the process which executes the operation and thus influences the scheduling of the underlying operating system.

Execution intervals can be deduced from assignment and priority of operations assuming that always the executable operation (i.e. all operands are at least partially available) with highest priority is scheduled.

The number of possible parallel execution plans is much larger than the number of single processor execution plans due to the possibility of node splitting and the scheduling decisions. Hence, while the complete enumeration of all reasonable single processor execution plans may still be feasible, this is certainly impossible for parallel execution plans. Therefore, we need effective heuristics to prune the search space.

Former solutions for the generation of parallel execution plans are not convincing. [CePS85] considers only parallelism between independent operations and an a priori fixed degree of node splitting for each operation. [BaYH87] assumes that node splitting is always possible without additional CPU-cost, and uses pipelining always, if any reduction in execution time is achieved, thereby possibly precluding a better parallelization. Hence we have

Problem 5: *What are effective heuristics for the generation of parallel execution plans?*

We want to find execution plans which maximize the degree of parallelism subject to a maximum x%-increase in total processing cost. Two basic decisions are involved in this task: adding additional potential parallelism by node splitting and realising parallelism by associating processors and priorities with operations. As we can not decide in advance, wether it is better to use pipelining or node splitting, we define potential parallelism on a per *pipe* basis (a pipe is built by several operations (possibly only one) communicating with each other in a pipelined fashion). Thus the generation of parallel execution plans also involves a number of steps:

(1) Iteratively generate execution plans with increased potential parallelism (number of processors associated with a pipe). The search can be limited, as in each iteration we only have to increase the number of processors for pipes lying on the critical path in order to obtain reduced execution times.

(2) Minimize the execution time of a pipe utilizing the number of processors allowed. We may use node splitting and have to assign operations to processors. The execution time is minimized by balancing

the load across the participating processors with as less expensive node splitting and communication as possible.

(3) Determine an optimal schedule given the degree of parallelism and execution time of each pipe. The scheduling decisions can be based on a heuristic which has been proven to be useful in deterministic scheduling theory: critical path scheduling.

We expect that both degree of parallelism and processing cost will increase while we proceed increasing potential parallelism, hence we can finally choose an optimal processing plan for each increase in total cost we are interested in.

6 Conclusion

We have described an approach to the optimization of SQL queries for parallel execution in a multiprocessor database machine, which, considering the increasing performance of communication networks, will become relevant for distributed database systems as well. We plan to implement it until the end of this year, thus replacing the limited optimizer for an SQL subset currently used within the prototype of the KARDAMOM database machine.

Our approach is rather static in that only a limited number of decisions is performed at runtime. Therefore it depends on reliable cost estimates which are not easy to obtain. Thus the next problem to be examined will be

Problem 6: *What is the influence of imprecise cost estimates on the quality of execution plans? In case imprecise estimates have a significant influence, how can we perform dynamic optimization at runtime?*

An approach to solve the first part of this problem is to use our optimizer to produce execution plans under different cost estimates and compare their quality under the different estimates. The design of dynamic optimization at runtime can be based upon an analysis of the differences of the produced execution plans.

References

[AlCo88] W. Alexander, G. Copeland: Process and Dataflow Control in Distributed Data-Intensive Systems. Proc. ACM SIGMOD, Chicago, June 1988

[BaYH87] T. Baba, S.B. Yao, A.R. Hevner: Design of a Functionally Distributed Multiprocessor Database Machine Using Data Flow Analysis. IEEE Trans. on Computers, C-36,6, June 1987, pp. 650-666

[BoDe82] H. Boral, D.J. DeWitt: Applying Data Flow Techniques to Database Machines. IEEE Computer, August 1982, pp. 57–63

[Bült87] G. v. Bültzingsloewen: Translating and Optimizing SQL Queries Having Aggregates. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, September 1987, pp. 235-243

[Bült89] G. v. Bültzingsloewen, C. Iochpe, R.-P. Liedtke, R. Kramer, M. Schryro, K. R. Dittrich, P. C. Lockemann: Design and Implementation of KARDAMOM — A Set-oriented Data Flow Database Machine. To appear, 6th Int. Workshop on Database Machines, Deauville, June 1989

[CABK88] G. Copeland, W. Alexander, E. Boughter, T. Keller: Data Placement in Bubba. Proc. ACM SIGMOD, Chicago, June 1988, pp. 99-108

[CeGo85] S. Ceri, G. Gottlob: Translating SQL into Relational Algebra: Optimization, Semantics and Equivalence of SQL Queries. IEEE Trans. S.E., April 1985, pp. 324-345

[CePS85] F. Cesarini, F. Pippolini, G. Soda: A Technique for Analyzing Query Execution in a Multiprocessor Database Machine. Proc. 4th Int. Workshop on Database Machines, Grand Bahama Island, March 1985, pp. 68-90

[Chan76] P.Y. Chang: Parallel Processing and Data Driven Implementation of a Relational Database System. Proc. of the 1976 Conf. of the ACM, pp. 314-318

[Coff76] E.G. Coffmann (ed.): Computer and Job-Shop Scheduling Theory. John Wiley & Sons, 1976

[Daya87] U. Dayal: Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, September 1987, pp. 197-208

[DeGe85] D.J. DeWitt, R. Gerber: Multiprocessor Hash-Based Join Algorithms. Proc. 11^{th} Int. Conf. on Very Large Data Bases, Stockholm, 1985

[DeWi86] D.J. DeWitt et al.: GAMMA - A High Performance Dataflow Database Machine. Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, August 1986

[GaJo79] M.R Garey, D.S. Johnson: Computers and Intractability — A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, San Francisco, 1979

[GrDe87] G, Graefe, D.J. DeWitt: The Exodus Optimizer Generator. Proc. ACM SIGMOD, San Francisco, May 1987, pp. 160-172

[IoWo87] Y.E. Ioannidis, E. Wong: Query Optimization by Simulated Annealing. Proc. ACM SIGMOD, San Francisco, May 1987, pp. 9-22

[JaKo84] M. Jarke, J. Koch: Query Optimization in Database Systems. ACM Computing Surveys, June 1984, pp. 111-152

[Kim 82] W. Kim: On Optimizing an SQL-like Nested Query. ACM TODS, Sept. 1982, pp. 443-469

[Klug82] A. Klug: Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. Journal of the ACM, Vol. 29, No.3, July 1982, pp. 699-717

[KTMo83] M. Kitsuregawa, H. Tanaka, T. Moto-oka: Application of Hash To Data Base Machine and Its Architecture. New Generation Computing, Vol. 1, No. 1, 1983

[LeFL88] M.K. Lee, J.C. Freytag, G.M. Lohman: Implementing an Interpreter for Functional Rules in a Query Optimizer. IBM Research Report RJ 6125, March 1988

[LeVi85] C. Le Viet: Translation and Compatibility of SQL and QUEL Queries. Journ. Inf. Proc., Vol. 8, No. 1, 1985, pp. 1-15

[Lori89] R. Lorie et. al.: Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience. IEEE Data Engineering, Vol. 12, No. 1, March 1989, pp. 2-8

[Pear84] J. Pearl: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison Wesley, 1984

[RiLM87] J.P. Richardson, H. Lu, K. Mikkilineni: Design and Evaluation of Parallel Pipelined Join Algorithms. Proc. ACM SIGMOD, San Francisco, May 1987, pp. 399-409

[Seli79] P.G Selinger et. al.: Access Path Selection in a Relational Database System. Proc. ACM SIGMOD, Boston, May 1979, pp 23-34

[Shan88] M.-C. Shan: Optimal Plan Search in a Rule-Based Query Optimizer. Proc. Int. Conf. on Extending Database Technology, Venedig, March 1988, pp. 92-112

[SwGu88] A. Swami, A. Gupta: Optimization of Large Join Queries. Proc. ACM SIGMOD, Chicago, June 1988, pp. 8-17

Communication Considerations for Query Optimization for Distributed/Parallel Systems[1]

(Extended Abstract)

W.S. Luk and Xiao Wang
School of Computing Science
Simon Fraser University
Burnaby, B.C. Canada V5A 1S6
(E-mail: woshun@cs.sfu.ca)

1. Introduction

A distributed database system is distinguished from a centralized one by the fact that the database in the former is distributed geographically over more than one site. Consequently, any scheme to evaluate distributed query must consider the cost of moving data from one site to another (i.e. communication cost), in addition to the processing cost. The main difficulty one encounters in estimating the cost of processing a distributed query is that the communication cost and processing cost are often evaluated in different and incompatible units.

In the literature, there are many different approaches to arriving at a solution to this problem. The most common one is perhaps the simplest one: the processing cost is considered to be negligible in comparison with the communication cost. The cost formula is usually a linear function of the amount of data transmitted. In many research papers, the startup cost (i.e. the constant in the linear function) is considered to be zero, so that the communication cost is equivalent to the amount of data transmitted. A good survey of research work adopting this approach may be found in [YC 84]. A variation of this approach is to assume that local processing has taken place before data transmission ([GS 86]).

In some systems, all three costs, i.e. disk retrieval cost, communication cost and processing (or CPU) cost are considered. The cost formula is a linear function of these three cost components with an appropriate weight factor assigned to each of them. System R* adopted this approach ([SA 80]).

Recently, there have been several experimental studies in evaluating various distributed query processing strategies (e.g. [HF 86] and [CAR 85]). These strategies were actually implemented on a distributed system and the actual timing information of each strategy was collected for comparison. These systems are often disk-based, so that disk I/O cost accounts for a substantial portion of total elapsed time.

Like [HF 86], we believe that distributed query optimization can best be studied by observing an actual system, since the interrelationships among the application software, underlying distributed operating system and communication hardware are quite complex. On the other hand, we also believe that the analytic modelling approach will continue to be usefully employed because this low-cost approach *can* provide a rough estimate of the system performance so that obviously poor strategies may be ruled out before the implementation phase begins. Furthermore, it is our experience that faulty implementations can be detected by means of analytic techniques.

Over the last two years, we have been studying the communication aspect of distributed query optimization by actually implementing distributed algorithms in a network of Sun workstations over an Ethernet local area network. Specifically, the disk I/O aspect of the algorithms is not considered, and no program has to fetch data from the disk. Three different types of algorithms have been considered:

---

[1]This work was partially supported by grants from the Natural Sciences and Engineering Research Council of Canada and the Centre for Systems Science at Simon Fraser University.

distributed sorting algorithms, distributed join algorithms and transitive closure algorithms. Typically, for each type of algorithms, we implemented selective algorithms published in the literature and our own algorithms, which were devised based on the observations of the experimental results of these algorithms. These algorithms were evaluated on the basis of the total response time, i.e. the longest wall-clock time of all machines participating in the distributed operation. Specific results related to each class of algorithms have been reported in [LL 89], [WL 88] and [A 88]. In this paper, we provide a summary of our observations regarding the communication aspect of these algorithms, which we believe are applicable to distributed query optimization in general.

2. Observations

2.1. The Hardware/Software Factors

The "raw" communication performance, i.e. the data transfer speed between two applications running in different sites at the operating system level, is a critical factor in the design of a distributed database system. Given that the bandwidth of the Ethernet is fixed, two main factors that may affect the communication performance are: the underlying distributed operating system and the main processor in each site. The following table shows the performance of the V-System[2] in data transfer between two diskless Sun-2's. In each case, a message of a certain size is transmitted from one site to another and communication time (cost) is taken to be the time between initiation of the send command and reception of the acknowledgement from the receiver.

32 bytes	0.5K bytes	1K bytes	2K bytes	4K bytes	8K bytes	16K bytes
3.1 ms	6 ms	8.6 ms	16.4 ms	23.9 ms	38.5 ms	68.5 ms

Table 2-1: SUN-2 V-System IPC Timing Information

For the identical system configuration, the performance for Berkeley Unix 4.2 is a lot worse. Note that in the case of Berkeley Unix 4.2, TCP/IP was employed. The typical overhead of creating a TCP/IP session between any two sites, which ranges from 220 and 400 ms, was not included in the following table. Thus in actual application environment, the communication performance of Berkeley Unix 4.2 will be even worse.

32 bytes	0.5K bytes	1K bytes	2K bytes	4K bytes	8K bytes	16K bytes
6.7 ms	15.7 ms	17.6 ms	26.6 ms	54 ms	109 ms	220 ms

Table 2-2: SUN-2 BSD 4.2 IPC Timing Information

A fast processor will definitely have great impact on the communication performance. The following table shows the communication performance of Sun-3 on the V-System.

32 bytes	0.5K bytes	1K bytes	2K bytes	4K bytes	8K bytes	16K bytes
2 ms	4 ms	6 ms	9 ms	11 ms	16 ms	25 ms

Table 2-3: SUN-3 V-System IPC Timing Information

[2]The V-System is a message-based, high-performance distributed operating system that runs on Sun and VaxStation workstations [Che 84]. While it is intended to be a general-purpose operating system, it also has a low operating system overhead which implies a low 'noise' level when the performance of the application is measured. In addition, it has a very finely tuned interprocessor communication mechanism.

2.2. Short Messages vs. Long Messages

The first observation we can make from the above tables is that the time required to transmit short messages, i.e. 32-byte messages is not much shorter than the time required to transmit much longer messages[3]. Indeed, the unit transmission time (i.e. time to transmit one byte) is 40 times larger for a short message than that for a long message (Table 2-3).

There are two consequences arising from this observation. First, it is necessary to provide different time measures for transmissions of short and long messages. In [LL 89], the time for transmission of short messages is measured by the number of messages, while the time for transmission of long messages is measured by the total amount of data for transmission. The resulting timing analysis was confirmed by empirical data. Second, it is a common practice to assume that data will be transmitted as soon as they are ready. However, batching of several short messages for one transmission is beneficial so long as the generation of one short message is not dependent on the transmission of an earlier short message.

2.3. Impact of Processor Speed on Communication Time

Due to protocol processing by the processor, the network engagement time, which is the amount of time required for data to travel from one site to another site over the network medium, may account for a very small percentage of the total transfer time (i.e. communication time in this paper). Take for example the transfer of a 1K-byte message in a BSD 4.2 system (see Table 2-2). The network engagement time for this message over a 10Mbit/sec Ethernet accounts for 2.5% of the total time. Nonetheless, this percentage value depends on the operating system, processor speed and the network bandwidth. For example, this percentage becomes considerably higher for SUN-3 over the V-System. The transfer rate for 16K-byte messages is about 5Mbit/sec. This means that the network engagement time accounts for about 50% of the total time.

This observation will have significant implications for optimization of communication cost, as the following two subsections show.

2.4. Concurrent Transmissions

Most distributed query processing algorithms we have implemented proceed by stages such that at the end of each stage, the machines (Sun workstations) exchange data with each other at roughly the same time. Consequently it is important to schedule concurrent transmissions properly.

Generally speaking, concurrent transmissions should be scheduled sequentially to avoid excessive packet collisions and subsequent retransmissions. However, when the percentage of network engagement time over the total communication time is small, one can take advantage of *communication parallelism* by allowing concurrent transmissions to proceed asynchronously. In this way, the time for concurrent transmissions would be drastically reduced since the protocol processing by each individual machine may proceed in parallel.

2.5. Network Bandwidth

With microprocessors becoming more powerful all the time, the network bandwidth is a potential bottleneck in a distributed database system. Already the network engagement time for a Sun3-to-Sun3 data transfer accounts for 50% of the total transfer time.

Conversely, if the network bandwidth is increased by 10 folds, the percentage of the network engagement time out of the total time will shrink from 50% to 10%. Communication parallelism would be beneficial again.

---

[3]This of course is mainly due to the fact that the maximum packet size in an Ethernet network is 1K-byte.

2.6. Communication/Local-Processing Trade-off

The assumption that communication is the dominant cost factor in distributed database processing is not borne out by the performance data we have collected. We suspect the assumption is generally not valid for distributed database processing over local area networks. In fact, it is often a good policy to distribute the workload to keep the machines busy at the expense of extra communication overhead. In [WL 88], we showed that the operation of building a hash table can be effectively distributed to 4 machines.

2.7. Parallel vs. Distributed Database Processing

While the system in which the experiments were performed is definitely a distributed system, our algorithms can also run on a loosely coupled (i.e. distributed memory or message-passing) multiprocessor system with only cosmetic changes. Naturally the bus speed is much higher than that on Ethernet. Perhaps the penalty for sending short messages on a bus is not as severe. Despite the speed limitation of the Ethernet, many (but not all) distributed sorting, distributed join and distributed transitive closure algorithms have achieved speedups over a network of Sun workstations in comparison with their uni-processor counterparts.

The basic strategy adopted to achieve maximum speedups is to keep all processors busy by distributing data to them. Thus the communication cost is the overhead for parallel database processing. As the number of machines increases and the communication cost will not decease, there is a limit to the number of machines that can be usefully employed for parallel processing in our processing configuration, given of course, a fixed file size. In all three classes of algorithms we have considered, the optimal number is no more than sixteen, given a global file size of 64K integers (or tuples).

Interestingly, those distributed sorting algorithms that were adapted from parallel sorting algorithms intended for a processor array do not perform as well as other algorithms intended for a distributed system [LL 89]. This is due to the fact that the grain size assumed by the parallel sorting algorithms is too fine for the message-passing architecture.

2.8. Broadcasting

Many distributed database operations involve broadcasting messages/files to all other machines on the network. Since Ethernet is inherently a broadcast network, it is very tempting to assume the time of broadcasting one message is equivalent to the time required for a *point-to-point* transmission of that message.

In actual fact, it is not so. We have found that for both BSD 4.2 Unix and V-System, it is more efficient to do multiple *point-to-point* transmissions than one broadcast. This is due to the difficulty of handling multiple acknowledgements that arrive almost simultaneously at the Ethernet controller of the broadcaster. We are currently working at some schemes to modify the system kernel of V-System to make broadcasting work.

2.9. Future Research Directions

It appears that high bandwidth networks (>= 100 Mbps) are becoming available soon. Coupled with a high speed microprocessor, the memory-to-memory data transfer will be much speedier. Also, as the high-density memory chips (4 to 16 Mbits) become commercially available, the memory cost will drop, and the disk I/O factor in query optimization will be less important. All these should have a significant impact on the design of distributed query processing algorithms.

With a much lower overhead in terms of communication, parallel database processing will become more widespread. From the research perspective, what is the best parallel database architecture? Bus (Ethernet-like), hypercube or tree?

3. References

[A 88] Almstrom, C.
Distributed Computation of Transitive Closure.
Master's thesis, Simon Fraser University, Burnaby, B.C., Canada, 1988.

[CAR 85] Carey,M.J. & Lu,H.
Some Experimental Results on Distributed Join Algorithms in a Local Network.
Technical Report 587, Comp Sc Dept, U. of Wisconsin-Madison,March, 1985.

[Che 84] Cheriton, D.R.
The V Kernel: A Software Base for Distributed Systems.
IEEE Software. 1(2), April, 1984.

[GS 86] Gavish, B. and Segev, A.
Set Query Optimization in Distributed Database Systems.
ACM Trans. on Database Systems 11(3), September, 1986.

[HF 86] Hagmann, R.B. and Ferrari, D.
Performance Analysis of Several Back-End Database Architectures.
ACM Trans. on Database Systems 11(1), March, 1986.

[LL 89] Luk, W.S. & Ling, F.
An Analytic/Empirical Study of Distributed Sorting on a Local Area Network.
IEEE Transactions on Software Engineering 15, May, 1989.

[SA 80] Selinger, P.G. and Adiba, M.E.
Access Path Selection in Distributed Database Management Systems.
In *Proc. of the Inter. Conference on Databases.* July, 1980.

[WL 88] Wang, X. and Luk, W.S.
Parallel Join Algorithms on a Network of Workstations.
In *Inter. Symp. on Databases in Parallel and Distributed Systems.* IEEE Computer Society, December, 1988.

[YC 84] Yu, C.T. and Chang, C.C.
Distributed Query Processing.
ACM Computing Surveys 16(4), December, 1984.

Site Selection in Distributed Query Processing

T. Patrick Martin

Department of Computing & Information Science
Queen's University
Kingston, Ontario K7L 3N6.

(613) 545-6063

E-mail: martin@qucis.queensu.ca

1. INTRODUCTION

A key component of any relational database system is the query processing subsystem. It is vital to the performance of the system that the most efficient version of a user's query is produced before the query is handed over for processing. This is especially true in a widely-distributed database system where just the communication costs alone can be prohibitively expensive.

We consider the processing of a distributed query to be divided into three phases: the *site selection* or *copy identification* phase, the *reduction* phase and the *assembly* phase [1]. In the site selection phase, one or more copies of each relation in the query are chosen. In the reduction phase, semijoins are normally used to eliminate tuples of the relations that do not satisfy the qualification of the query. In the assembly phase, relations in the qualification component of the query are sent to one site to produce the result.

Site selection plays a critical role in query optimization for replicated, widely-distributed databases. Despite this fact, it has received little attention compared with the reduction and assembly phases and, in practice, is often bypassed or assumed to be solvable by simple enumeration. This assumption only holds for very simple queries involving a small number of relations and, in general, the problem of finding the allocation of subqueries to sites that yields the minimum cost is an NP-hard problem [2].

The main goal of our project is to develop cost-effective methods of site selection for query processing in large, heterogeneous, widely-distributed databases. There are a number of tasks involved in achieving this goal: defining a representative cost model; choosing heuristics and algorithms which will make the site selection process computationally feasible; incorporating both static and dynamic system information into the decision process, and understanding the effects different properties of the site selection problem will have on the results produced for each of the preceding tasks.

2. WORK IN PROGRESS

Initially, we are considering a homogeneous distributed database and queries that are run repeatedly, that is, queries that are *compiled* and the resulting access plans saved. We are experimenting with a two stage approach to site selection. The first stage, called *static selection*, takes the optimization process as far as it can using only static information about the database and the system, and a description of the "typical" state of the system. Static information includes profiles of the relations and attributes, the available implementations for the different relational operators at the sites, the capacity, processing rate and I/O rate of each the sites, and the data rates of the paths between pairs of sites. The typical state of a system is represented by all sites and links experiencing their average load and the base relations being used by their average number of concurrent

This work is supported by the Natural Science and Engineering Research Council of Canada under grant OGP0000929.

transactions. These average values have to be arrived at by observing the system. The state space of possible allocations can be pruned significantly using just static information and, depending upon the closeness of the system state at execution time to the typical state, further optimization may not even be required at runtime. The cost of static site selection is amortized over the repeated executions of the query.

The second stage, called *dynamic selection*, is employed at query execution time if the static allocation is rendered infeasible because the current system state deviates from the assumed typical state as the result of failures or heavy loads on particular sites or links. Dynamic selection must find an acceptable alternative in a small amount of time since it directly results in a delay of query execution. Dynamic selection would also be used for ad-hoc, or one-time, queries where static selection is not appropriate.

We believe that a two stage approach to query optimization is required in a large widely-distributed database system. Dynamic optimization alone, while suitable for a local area network environment [3,4], is not enough in our environment. The number of alternative access plans is extremely large for even relatively simple queries and our experiments have shown that fast heuristic algorithms cannot find acceptable near-optimal solutions in these large search spaces. Static optimization can narrow the search space so that these fast algorithms have a much better chance of finding a good access plan. Similarly, just static optimization is not sufficient since the system state at runtime may be radically different from the state at optimization, for example some of the sites could be down. The access plan has to be adjusted, or a new plan found, if the query is to be executed.

We conducted a series of experiments to analyze the performance of a number of different types of algorithms for finding a near-optimal allocation of subqueries to sites during the static selection stage [5]. We considered five different types of algorithms - branch-and-bound (BB), greedy (GR), iterative improvement (II), local search (LS) and simulated annealing (SA). The experiments used a set of queries ranging from simple to very complex and assumed a database consisting of a large number of relations and/or fragments. Both full and partial replication of the relations were considered. The algorithms were evaluated based on the communication costs involved in their solutions, and the runtimes required to reach those solutions.

The BB and GR algorithms were found to perform well for simple queries. The BB algorithm's runtime grew exponentially once complexity rose to a moderate level which makes this class of algorithm of limited use in a large distributed database system. The GR algorithm gave a low cost solution only so long as all of the processing could be done at a single node. This is not practical in a partially-replicated system and eliminates the GR algorithm from serious consideration. The II algorithm required relatively little runtime but did not produce a satisfactory query cost for any of the queries. II would only be useful in the case where the amount of runtime was very limited and the query was too complex to be answered by an enumerative algorithm such as BB.

The LS and SA algorithms both yielded reasonable solutions for the whole range of queries but at significant runtime costs, especially for the more complex queries. The LS algorithm always found a better query cost than SA for both the partially-replicated and the fully-replicated databases. Our experiments demonstrated the cost-effectiveness of sophisticated algorithms, such as LS and SA, for static site selection when we are dealing with compiled queries in a large widely-distributed database system. The other simpler algorithms could not find suitable allocations in at least some subset of the queries examined.

Our results indicate that no one algorithm is dominant for all cases. The choice of the algorithm must be made based on the complexity of the query and the environment, and on the importance of obtaining a good query cost. An enumerative algorithm is best for simple queries. The point at which the runtime becomes unreasonable depends upon query complexity and the size of the system. Local search is the best approach for complex queries. It yields a good query cost in all cases but is not cost-effective for the simpler queries relative to an enumerative algorithm.

Our findings seem contrary to those of Bodorik and Riordon [6] who favour a greedy algorithm for static access plan selection in a distributed database. The different conclusions stem from

different assumptions about the system model and a different objective function. Our experiments assumed a much larger system than those reported by Bodorik and Riordon. Also, their experiments were concerned with response time, not total query cost, so algorithm runtime played a more important role in their evaluation.

3. FUTURE WORK

We are currently trying to gain more insights into the use of the local search algorithm and into the properties of the site selection problem by conducting experiments with local search over a wide range of query complexities and system sizes. We hope to analyze the effect different parameters of the problem have on the performance of the local search. We are also investigating variations of the algorithm. Local search is very dependent upon the choice of an initial allocation and an efficient way of narrowing the options should reduce the computation time required to arrive at a good solution. One variation is to use a greedy algorithm to quickly arrive at a potentially good initial allocation and then refine that allocation using local search. A second variation is a distributed version of the algorithm. Local search lends itself to distribution so there is a possibility of significant runtime savings.

The problem of developing appropriate cost models is vital to research in distributed query optimization. The cost model we used for our experiments, taken from the work by Liu and Chang [7], was too abstract. It considered only select, project and join operators and made a number of other simplifying assumptions. We hope to develop a more detailed cost model that, besides the select, project and join operators, will also include the union and semijoin operators; that will consider alternative implementations of operations, and that will allow other distributions, besides the uniform distribution, for attribute values.

An extension of a transformation-based approach, such as the one described by Rosenthal and Helman[8], seems promising for two reasons. First, the operator graphs of Rosenthal and Helman already contain much of the static information required; in particular, they can represent alternative strategies for performing an operation. We would like to extend the graphs to also incorporate the dynamic system state information. Second, the transformation-based approach is appropriate for a heterogeneous environment. Operators can be initially expressed at a high level and then refined in different ways depending upon the type of database chosen to answer the subquery.

We also plan to study the impact of changing the objective function from minimizing total query cost to minimizing response time. Perhaps different types of queries respond better to one or the other objective function. It would also be interesting to see what effect a mixture would have on overall system performance.

The two main areas we wish to study with regards to dynamic selection are the algorithms that could be used, and the definition, representation and maintenance of system load. The main characteristic of an algorithm used for dynamic selection is that it should arrive at a reasonable allocation of subqueries to sites in a minimum amount of time. Algorithm runtime is important since it will be a component of the delay the query experiences before it can execute. Our study of algorithms for the static case suggests that greedy or iterative improvement algorithms are the most likely candidates.

A key concept in dynamic selection is that the algorithm should make use of as much of the information provided by the static selection as possible. One approach could be to salvage at least a partial solution from the static allocation and then to converge to a total allocation from that point. A second approach could be to have the static selection produce several candidate access plans and the dynamic selection could refine these based on current state information and choose the best one. If none of the candidate plans could be executed in the current state we would have to resort to the first approach.

The problem of defining system load can be approached in several ways depending upon the assumptions made about the database system. For very large homogeneous distributed databases communication costs will dominate. Copies of a relation are likely to be far apart and choosing one

copy over another would involve significant communication cost differences that probably would outweigh any differences caused by delays at sites. Thus only site failures or substantial congestion on a path would force a change in the allocation. The system state, in this case, could be represented by a flag for each site indicating whether that site is up or down, and a load measure, such as message queue length, for each path between pairs of nodes. We assume that this load measure is for the "best" path between the nodes at that time if there are multiple paths.

If we assume a heterogeneous system, with respect to sites, links and even databases, or if we assume the use of a high bandwidth transmission technology such as fiber optics, then differences in processing costs and delays will be significant. For example, performing the join of two large relations on a close PC is likely to be more costly than performing the same join on a more remote mainframe. In this case some measure of the load at a site is required such as CPU queue length, CPU utilization, estimated response time or the number of CPU-bound and I/O-bound tasks. While we are not sure if such a level of detail is required, it would be theoretically interesting to try and include some measure of the load on the any copies of base relations at a site. For example, the number and types of locks held on a copy of a relation give an indication of the load on that copy.

4. SUMMARY

We are studying the problem of site selection in large heterogeneous distributed databases. We are experimenting with a two stage approach to the problem which first finds a near-optimal allocation of subqueries to sites using the static information available about the database and the network, and second, if necessary, revises the allocation based on the current state of the system. We feel that the three most important tasks in this research are the development of an appropriate cost model, the development of algorithms and heuristics to carry out the site selection at an acceptable time cost, and the refinement of the concept of dynamic selection. Although dynamic selection in widely-distributed databases has not received very much attention until now, its use could result in substantial improvements in the reliability of the query processor.

The general area of distributed query processing, and the particular problem of site selection, for large heterogeneous distributed databases still require a great deal of study. While we can use some of the techniques from related areas such as query processing for centralized databases and task allocation in distributed systems, unique aspects of the problem require new solutions. This search for solutions is complicated by the lack of working examples of such systems which forces us to hypothesize about the characteristics of these databases and about their workloads.

5. REFERENCES

[1] C.T. Yu and C.C. Chang, "Distributed Query Processing", *ACM Computing Surveys 16(4)*, December 1984, pp.399-433.

[2] C.T. Yu, C. Chang and Y. Chang, "Two Surprising Results in Processing Simple Queries in Distributed Databases", *Proceedings of the IEEE 6th International Computer Software and Application Conference*, Chicago, 1982, pp.377-384.

[3] M.J. Carey and H. Lu, "Load Balancing in a Locally Distributed Database System", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, May 1986, pp.108-119.

[4] P. Agrawal, D. Bitton, K. Guh, C. Liu and C. Yu, "A Case Study for Distributed Query Processing", *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, Austin, December 1988, pp.124-130.

[5] T.P. Martin and K.H. Lam, "Algorithms for Site Selection in Distributed Query Processing", Technical Report 88-240, Dept. of Computing and Information Science, Queen's University, Kingston, November 1988, and submitted to ACM-SIGMOD 1989.

[6] P. Bodorik and J.S. Riordon, "Heuristic Algorithms for Distributed Query Processing", *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*,

Austin, December 1988, pp.144-155.

[7] A.C. Liu and S.K. Chang, "Site Selection in Distributed Query Processing", *Proceedings of the 3rd International Conference on Distributed Computer Systems*, Miami, October 1982, pp.7-12.

[8] A. Rosenthal and P. Helman, "Understanding and Extending Transformation-Based Optimizers", *Database Engineering 9(4)*, December 1986, pp.44-51.

Query Optimization in a Main-Memory-Resident Database System[1]

Kyu-Young Whang

IBM Thomas J. Watson Research Center
P. O. Box 218
Yorktown Heights, New York 10598
CSNET: WHANG@IBM.COM

Ravi Krishnamurthy

M.C.C.
3500 West Balcones Center Dr.
Austin, TX 78759
CSNET: RAVI@MCC.COM

We present techniques for optimizing queries in memory-resident database systems. We discuss them in the context of Office-by-Example(OBE) that has been under development at IBM Research. Optimization techniques in memory-resident database systems differ significantly from those in conventional disk-resident database systems. In particular, the following aspects are of importance:

1. A new approach to developing a CPU-intensive cost model.
2. New optimization strategies for main-memory query processing.
3. New insight into join algorithms and access structures that take advantage of memory-residency of data.
4. The effect of the operating system's scheduling algorithm on the memory-residency of data.

[1] The full paper will appear in the ACM Transactions on Database Systems.

We present a simple index data structure suitable for a memory-resident database. The index is implemented as a flat array of TIDs that are pointers to tuples. This structure saves the storage space significantly compared with conventional index structures. The reduction of the storage space allows us to have more indexes with less storage overhead. In fact, in OBE, it is possible to implement the strategy of having indexes for all the attributes in the database. This strategy obviates physical database design problem, which is a nuisance for novice users.

We emphasize that a proper scheduling algorithm of the operating system is crucial for realizing a memory-resident database system. In particular, we show that the working-set scheduling algorithm provides an excellent approximation for memory-residency of data. By using this algorithm, the system prevents potential thrashing due to heavy usage of virtual memory. In contrast, a pure demand paging scheme would not work in a practical time-shared environment (even with a physical memory size sufficient for a single user) because of potential thrashing.

Finally, we believe that a database system based on the memory-residency assumption is suitable for efficient main-memory applications including many aspects of artificial intelligence and logic programming (such as Prolog [9]). In particular, nonrecursive queries expressed in function-free Horn-clause logic can be directly processed by the techniques proposed in this paper [43].

13. Hall, P. A. V., "Optimization of a Single Relational Expression in a Relational Database," *IBM J. of Res. and Dev.*, Vol. 20, No. 3, pp.244-257, 1976.

14. Hammer, M. and Chan, A., "Index Selection in a Self-Adaptive Database Management System," In *Proc. ACM Intl. Conf. on Management of Data*, Washington, D. C., pp. 1-8, June 1976.

15. Hiller, F. S. and Lieberman, G. J. *Introduction to Operations Research*, Holden-Day, Inc., San Francisco, CA, Third Edition, 1980.

16. Ibaraki, T. and Kameda, T., "On the Optimal Nesting Order for Computing N-Relational Joins," *ACM Trans. Database Syst.*, Vol. 9, No. 3, pp. 483-502, Sept. 1984.

17. IBM, *VM/SP: System Logic and Problem Determination Guide (CP)*, LY20-0892-2, Third Edition, IBM Marketing, Sept. 1983.

18. Jarke, M. and Koch, J., "Query Optimization in Database Systems," *ACM Computing Surveys*, Vol. 16, No. 2, pp. 111-152, June 1984.

19. Kambayashi, Y. and Yoshikawa, M., "Query Processing Utilizing Dependencies and Horizontal Decomposition," In *Proc. ACM Intl. Conf. on Management of Data*, San Jose, Calif., pp. 55-67, May 1983.

20. Kim, W., "On Optimizing a SQL-like Nested Query," *ACM Trans. Database Syst.* Vol. 7, No. 3, pp. 443-469, Sept. 1982.

21. Kitsuregawa, M. et al., "Application of Hash to Data Base Machine and its Architecture," *New Generation Computing*, No. 1, pp. 62-74, 1983.

22. Knuth, D., *The Art of Computer Programming—Sorting and Searching (Vol. 3)*, Addison-Wesley, 1973.

23. Kooi, R. and Frankforth, D., "Query Optimization in INGRES," *Database Engineering Bulletin*, Vol. 5, No. 3, IEEE Computer Society, pp. 2-5, Sept. 1982.

24. Krishnamurthy, R., Boral, H., Zaniolo, C., "Optimization of Nonrecursive Queries," In *Proc. 12th Intl. Conf. Very Large Data Bases*, Kyoto, Japan, pp. 128-137, 1986.

25. Lehman, T. and Carey, M., "Query Processing in Main Memory Database Management Systems," *Proc. ACM Intl. Conf. on Management of Data*, Washington, D.C., pp. 239-250, May 1986.

26. Lehman, T. and Carey, M., "A Study of Index Structures for Main Memory Database Management Systems," *Proc. 12th Intl. Conf. on Very Large Data Bases*, Kyoto, Japan, pp. 294-303, Sept. 1986.

27. Luk, W. S., "On Estimating Block Accesses in Database Organization," *Commun. ACM*, Vol. 26, No. 11, Nov. 1983.

28. Pecherer, R. M., "Efficient Evaluation of Expressions in a Relational Algebra," In *Proc. ACM Pacific Conf.*, pp. 44-49, 1975.

29. Power, L. R., "EPLEA, Using Execution Profiles to Analyze and Optimize Programs," IBM Res. Rep. RC9932, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, Apr. 1983.

30. Reiner, D. S., Ed., *Database Engineering Bulletin*, Vol. 5, No. 3, IEEE Tech. Comm. on Database Engineering, Sept. 1982.

31. Schkolnick, M. and Tiberio, P., "Estimating the Cost of Updates in a Relational Database," *ACM Trans. Database Syst.*, Vol. 10, No. 2, pp. 163-179, June 1985.

32. Selinger, P. G. et al., "Access Path Selection in a Relational Database Management System," In *Proc. ACM Intl. Conf. on Management of Data*, Boston, Mass., pp. 23-24, May 1979.

33. Shapiro, L., "Join Processing in Database Systems with Large Main Memories," *ACM Trans. Database Syst.*, Vol. 11, No. 3, pp. 239-264, Sept. 1986.

34. Smith, J. M. and Chang, P. Y., "Optimizing the Performance of a Relational Algebra Database Interface," *Commun. ACM*, Vol. 18, No. 10, pp. 568-579, 1975.

35. Stonebraker, M. et al., "The Design and Implementation of INGRES," *ACM Trans. Database Syst.*, Vol. 1, No. 3, pp. 189-222, Sept. 1976.

36. Ullman, J. D., *Principles of Database Systems*, Computer Science Press, Rockville, Maryland, 1982.

37. Vander Zanden, B.T., Taylor, H.M., and Bitton, D., "Estimating Block Accesses when Attributes are Correlated," *Proc. 12th Intl. Conf. Very Large Data Bases*, Kyoto, Japan, pp. 119-127, Aug. 1986.

38. Warren, D.H.D., "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," In *Proc. 7th Intl. Conf. Very Large Data Bases*, Cannes, France, pp. 272-281, 1981.

39. Whang, K.-Y., Wiederhold, G. and Sagalowicz, D., "Estimating Block Accesses in Database Organizations—A Closed Noniterative Formula," *Commun. ACM*, Vol. 26, No. 11, pp. 940-944, Nov. 1983.

40. Whang, K.-Y., Wiederhold, G. and Sagalowicz, D., "Separability—An Approach to Physical Database Design," *IEEE Trans. on Computers*, Vol. C-33, No. 3. pp. 209-222, Mar. 1984.

41. Whang, K.-Y., "Query Optimization in Office-by-Example," IBM Res. Rep. RC11571, IBM T.J. Watson Res. Center, Yorktown Heights, NY, Dec. 1985.

42. Whang, K.-Y. et al., "Office-by-Example: An Integrated Office System and Database Manager," *ACM Trans. Office Infor. Syst.*, Vol. 5, No. 4, pp. 393-427, Oct. 1987.

43. Whang, K.-Y. and Navathe, S., "An Extended Disjunctive Normal Form Approach for Processing Recursive Logic Queries in Loosely Coupled Environments," In *Proc. 13th Intl. Conf. Very Large Data Bases*, Brighton, England, pp. 275-287, Sept. 1987.

44. Wiederhold, G., *Database Design*, McGraw-Hill, New York, 1983.

45. Winston, P.H., *Artificial Intelligence*, Addson-Wesley, 1979.

46. Wong, E. and Youseffi, K., "Decomposition—A Strategy for Query Processing," *ACM Trans. Database Syst.*, Vol. 1, No. 3, pp. 223-241, Sept. 1976.

47. Yao, S. B., "Optimization of Query Evaluation Algorithm," *ACM Trans. Database Syst.*, Vol. 4, No. 2, pp. 133-155, June 1979.

48. Zloof, M. M., "Query-by-Example: A Data Base Language," *IBM Systems J.*, Vol. 16, No. 4, pp. 324-343, 1977.

49. Zloof, M. M., "QBE/OBE: A Language for Office and Business Automation," *IEEE Computer*, Vol. 14, No. 53, pp. 13-22, May 1981.

50. Zloof, M. M., "Office-by-Example: A Business Language that Unifies Data and Word Processing and Electronic Mail," *IBM Systems J.*, Vol. 21, No. 3, pp. 272-304, 1982.

Query Optimization for Memory-Resident Databases

(extended abstract)

Grant E. Weddell

Department of Computer Science, University of Waterloo
Waterloo, Canada, N2L 3G1
gweddell@uwaterloo.edu

1. PROJECT OVERVIEW

Almost any software system will have components that access and update information residing in main-memory. For example, a query optimizer for a database system has procedures to maintain the so-called "symbol table", which is essentially a memory-resident database of parsed queries, access strategy specifications, meta-data describing the various database schema, and so on. The Resident Database Manager (RDM) is the name of a software toolset under development at our institution that helps with the development of such components in a way similar to how Yacc, for example, helps with the development of other components responsible for parsing.

RDM presently consists of two compilers. Input to the first compiler is a list of specifications in an object-oriented database language called LDM, another acronym for "Logical Data Model". The main features of the various sublanguages of LDM are as follows:

- The data definition language manifests a data model that generalizes the relational model in two ways. First, the notions of "relation" and "domain" are combined into a notion of "class" by introducing surrogate keys for tuples, and by allowing attributes to be "tuple-valued". Second, classes can be organized in a generalization taxonomy whereby more specialized classes will automatically inherit attributes of more general classes. The taxonomy is established by declaring any number of immediate superclasses for each class (LDM supports so-called "multiple inheritance").
- Data access requests are expressed in a SQL-like query language that has been generalized for access to classes. Another component of the data manipulation sublanguage is a simple transaction language which allows the user to specify simple combinations of update operations on the database. One operation allows the user to change the "identity" of an object (effectively changing its type).
- A data statistics language (DSL) is used to specify statistical information about a database. Using the DSL, a user can supply estimates of the number of objects in a class, how often a query or transaction is invoked, the relative cost of space and time, and so on. The statistical information is used by the component of the compiler responsible for performance prediction.
- A storage definition language (SDL) can be used to override some of the decisions made by the compiler on internal encoding of data. In particular, a user can specify a selection of the indices to be maintained in order to support searching within the database, and a selection of storage managers for managing the space used by objects.

Output from the first compiler is access and representation code in a language called PDM (for "Physical Data Model"). PDM code can then be input to the second compiler along with applications written in an extended C language called C/DB. C/DB has additional language constructs that permit the direct use of data access specifications originally written in LDM. The result of this second compiler is "pure" C code that can then be compiled with a standard C compiler. A summary of overall dataflow for the toolset is given in Figure 1.

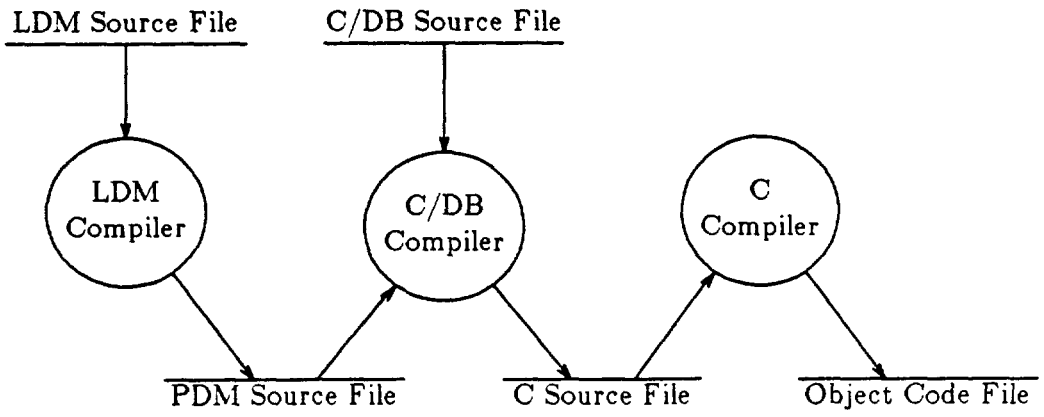


Figure 1. Basic Dataflow in RDM

2. OVERVIEW OF THE QUERY OPTIMIZER

Queries are represented internally by an access specification language (ASL).[4] The ASL is a wide-spectrum language in the sense that it allows the specification of a query with varying degrees of commitment to an access strategy. This includes forms that represent a number of alternative strategies.[3] Optimization is accomplished by a process of refinement, in which less committed ASL constructs are gradually replaced with more committed constructs that explicitly establish join order, index use, subquery organization, the order in which to check selection conditions, and so on. Each step in the refinement of a query corresponds to the application of a rewrite rule.[1] The possible sequencing of rules is expressed in a rule-control language, which also allows for explicit traversal of subparts of a query (in ASL).

An initial version of the query optimizer has now been completed in the Franz lisp programming language. At present, there are thirty-six rewrite rules used in essentially three separate phases of query optimization.

(a) *query normalization* — A "least committed" ASL version of the query is derived from its input form, which is specified in an SQL-like language.

(b) *join-order selection* — A nesting order for join operations is chosen, which also results in determining independent sub-queries. (Only the nested-loops strategy for join evaluation is expressible in our ASL. This is a reasonable limitation for memory-resident databases.[2,6]) Join order selection is accomplished by a branch-and-bound procedure derived from [5]. Note that the state of a branch-and-bound search is itself an ASL construct.

(c) *conjunct order selection* — Various post-join-order optimizations are performed. This includes the ordering of selections conditions within scans, automatic cut insertion and projection elimination. The latter optimizations use a procedure for reasoning about a form of functional dependency generalized for semantic data models.[7]

2.1. Example of Data Definition

In the next subsection, we illustrate operation of the query optimizer for an example query concerning a database of information about students, teachers and courses at some university. A description of the data is illustrated by the entity-relationship diagram in Figure 2. Specifications in the LDM data definition language that correspond to this description are given in Figure 3, with keywords in a bold typeface. Lines 36 to 40 express additional information in the LDM data statistics language that estimate the number of objects in various classes.

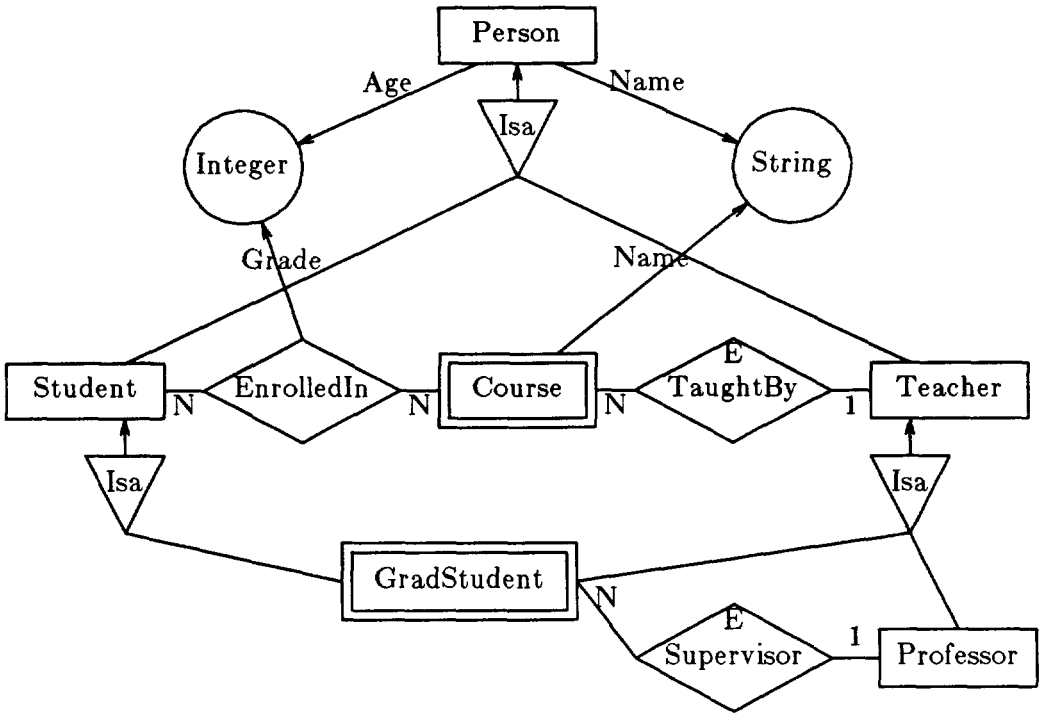


Figure 2. ER Diagram of a University Schema

2.2. Example of Query Optimization

In English, our example query is to

> "Find all graduates supervised by a professor who has taught a course in which a student with a given name has enrolled."

The query can be expressed in the LDM query language as follows.

```
1.  schema University
2.
3.  class Person
4.  properties Name, Age
5.  constraints
6.    Id determined by Name
7.    cover by Student, Teacher
8.
9.  property Name on String maxlen 20
10. property Age on Integer range 16 to 75
11.
12. class Student isa Person
13.
14. class Teacher isa Person
15. constraints cover by GradStudent, Professor
16.
17. class Professor isa Teacher
18.
19. class GradStudent isa Student, Teacher
20. properties Supervisor
21.
22. property Supervisor on Teacher
23.
24. class Course
25. properties Name, TaughtBy
26. constraints Id determined by Name
27.
28. property TaughtBy on Teacher
29.
30. class EnrolledIn
31. properties Student, Course, Grade
32. constraints Id determined by Student, Course
33.
34. property Grade on Integer range 0 to 100
35.
36. size Student 500
37. size GradStudent 100
38. size Course 100
39. size EnrolledIn 4000
40. size Professor 50
```

Figure 3. The University Schema in LDM

```
query SpecialGrads
given N from Name
select G from GradStudent
where exist P from Professor where
        G.Supervisor = P and
        EnrolledIn {N as Student.Name, P as Course.TaughtBy}
```

We shall illustrate the results of query optimization for SpecialGrads in two separate cases involving two different sets of index declarations in the LDM storage definition language.

Case 1. In this case, we assume four different indices have been declared by the user.

```
index PersonIndex on Person
of type binary tree ordered by Name asc

index GradIndex on GradStudent
of type binary tree ordered by Supervisor.Name asc, Age desc

index EList1 on EnrolledIn
of type distributed list on Course.TaughtBy

index EList2 on EnrolledIn
of type distributed list on Student
```

The first two indices are binary trees of all Person objects (including Student objects, and so on) and of all GradStudent objects respectively. In the first tree, the Person objects are sorted on the value of their Name property, and in the second tree on the value of the Name of their

Supervisor, then of their Age. The last two indices are essentially owner-coupled sets, with the class EnrolledIn as the member record type. EList1, for example, asserts that each Teacher object will point to a list of EnrolledIn objects for courses taught by the teacher.

The PDM specification indicating the access strategy generated by our query optimizer for this collection of indices is as follows.

```
query SpecialGrads
given N from Name
select G from GradStudent
declare S1, E1, P from Student, EnrolledIn, Professor
find all unique G
  subclass access S1 in PersonIndex with S1.Name = N;
  scan E1 in EList2 with E1.Student = S1;
  subclass assign P to E1.Course.TaughtBy;
  scan G in GradIndex with G.Supervisor.Name = P.Name;
  verify G.Supervisor = P;
  success
```

We refer to the "**find** ... **success**" construct used in the strategy as a *quant list*. A quant list represents a sequence of joins by nested iteration. This is illustrated by Figure 4, in which we present PASCAL-like code that would correspond to the above evaluation strategy for query SpecialGrads.

```
S1 := <first entry of PersonIndex where Name = N>;
if (not S1 = nil) and (S1.Name = N) and (S1 IN Student) then begin
  E1 := <first entry of EList2 where Student = S1>;
  while (not E1 = nil) and (E1.Student = S1) do begin
    P := E1.Course.TaughtBy;
    if (P IN Professor) then begin
      G := <first entry of GradIndex where Supervisor.Name = P.Name>;
      while (not G = nil) and (G.Supervisor.Name = P.Name) do begin
        if (G.Supervisor = P) then
          <remember G if not previously retrieved>;
        G := <next entry of GradIndex>
      end
    end;
    E1 := <next entry of EList2>
  end
end
```

Figure 4. Case 1 access strategy for query *SpecialGrads*

Note that "**unique** G" occurring after "**find**" indicates the need for a projection operation on GradStudent objects. This is necessary since a particular Professor object can be the Teacher value for more than one course for which a student with the given name is enrolled.

Case 2. Now consider where the user has declared only two indices which are lists of all Person objects and all EnrolledIn objects respectively.

index PersonList **on** Person **of type list**
index EList **on** EnrolledIn **of type list**

The access strategy generated by our query optimizer now has the following form.

query SpecialGrads
given N **from** Name
select G **from** GradStudent
declare E1, P **from** EnrolledIn, Professor
find all unique G
  scan E1 **in** EList;
  verify E1.Student.Name = N;
  subclass assign P **to** E1.Course.TaughtBy;
  subclass scan G **in** PersonList;
  verify G.Supervisor = P;
  success

This second strategy also has two loops, in which a scan of all Person objects using index PersonList is nested within a scan of all EnrolledIn objects using index EList. Note that a projection of graduate objects remains necessary for the same reason as in the first case above.

3. SUMMARY

We have found the selection of a richer object-oriented data model together with the physical circumstance of memory-resident to have a major impact on problems in query optimization.

- The index types appropriate in main-memory are not the same as traditional file structures used for disk-based storage of data.
- Clustering is much less of a performance issue. For example, the same record can serve as an index entry in more than one index.
- The circumstance of memory-residence together with the choice of an object-oriented data model introduces the possibility of sort conditions for indices that correspond to a "property path". An example of this is the "GradIndex" declared and used in Case 1 above.
- An object-oriented data model introduces the ability to express so-called. "functional joins". Memory-residence implies that functional joins can be supported for very little cost. Both cases above illustrate this.
- Cost models for performance prediction are different, and are also complicated by the presence of superclasses.

4. References

1. J. C. Freytag, A rule-based view of query optimization, *Proc. ACM SIGMOD Conference on Management of Data*, pp. 173-180 (May 1987).
2. R. Krishnamurthy and S. B. Navathe, A join processing strategy for memory-resident databases, Research Report, Microelectronics and Computer Technology Corporation (1987).
3. G. M. Lohman, Grammar-like functional rules for representing query optimization alternatives, *Proc. ACM SIGMOD Conference on Management of Data*, pp. 18-27 (June 1988).

4. R. A. Lorie and J. F. Nilsson, An access specification language for a relational data base system, *IBM Journal of Research and Development* **23**(3) pp. 286-298 (May 1979).

5. D. E. Smith and M. R. Genesereth, Ordering conjunctive queries, *Artificial Intelligence* **26**(1985).

6. G. E. Weddell, Physical design and query compilation for a semantic data model (assuming memory residence), Technical Report 198, Computer Systems Research Institute, University of Toronto (1987).

7. G. E. Weddell, Reasoning about functional dependencies generalized for semantic data models, Research Report CS-89-14, Department of Computer Science, University of Waterloo (1989).

A Performance Study of Nested Relational DBMSs involving Query Optimization

James E. Kirkpatrick and Mark A. Roth

Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433-6583

jkirkpat@afit.af.mil, mroth@afit.af.mil

March 31, 1989

1 Introduction

One of the reasons most often cited for studying the area of nested relations is the observed need for "better" support for advanced application areas whose requirements do not seem to be well served by the standard relational (or any other particular) data model. Those advanced application areas most often mentioned are: statistical data, VLSI CAD, engineering applications in general, and spatial/image data. A common theme is that these areas are not "simply" record-based such as the more traditional business/bank applications. They employ higher volumes of data within simple objects; the operations required do not match well with traditional DB operators (particularly relational algebra operators); the objects usually have inherent hierarchical relationships to one another which are not explicitly retained/expressed within a standard (relational) DB design; and ,in general, the requirements of the engineering/scientific community (among others) have progressed beyond the point where a standard relational DB, or any other existing type of DB, can provide adequate support.

One of the key elements of any "better" DBMS, however, must be its level of performance. No DBMS will be accepted, regardless of capabilities, if the resulting performance is unacceptable. Therefore, an interesting area of research might involve the investigation of the performance inherent in the new data models being developed. Nested relations provide one such data model.

The areas of nested relational database (NRDB) performance research to be found in the literature can be roughly categorized as:

- Proposals on how one might use the NR and complex object (CO) concept to implement a DB for a particular application (spatial, statistical, musical, etc) and why these models should be better (i.e., yield better performance) for the associated application.
- Proposals for Object Oriented (OO) DB models and Extensible DB systems which expect to result in better support of user requirements and performance increases. These proposals typically involve the concepts of complex objects, hierarchical decomposition, and inheritance which are/can be associated with a NRDB.
- Designs of actual NRDB implementation projects currently taking place (**DASDBS** [PSS+87], AIM [Dad88], and VERSO [Bid87]).
- Models/proposals for *parts* of a NRDB (i.e., storage methods [HO88a, HO88b], and query languages [RKB87]).

- Theoretical results which might be directly applicable in the development of NRDB performance models [Sch86, Bid87, DVG88].

We find it interesting that many papers begin with a section justifying the need for a NRDB model via examples of how NRs should be able to represent information more concisely and in a more logical (user-meaningful) manner than an equivalent series of flat relations. This section of a paper usually gives references to a series of example application areas where NRs should *intuitively* be a promising alternative to flat relations for data modeling. Unfortunately, we find no "real" justification for why the NR model should be strictly "better than" the flat model. In addition, there are no studies suggesting under which circumstances one model could be expected to lead to better performance than the other. It seems that this belief in a "better, more useful" NR model is common, but substantiated only in occasional and limited ways [Sch86, PSS+87].

At the Air Force Institute of Technology, we have begun a project involving the development of a NRDBMS using the EXODUS extensible DBMS. One of the primary purposes for initiating this project is to study performance aspects of nested relations. As a result, we have begun to investigate the optimization of queries within a NRDB.

In section 2, we briefly present some problems being addressed by our project as well as several of our research goals. Section 3 presents a quick look at concept of nested relations. Section 4 then gives an overview of query optimization results in the area of nested relations. Finally, section 5 presents a look at physical storage structures for nested relations.

2 AFIT NRDB Research Project

We believe that the "space" in which the performance of NRDBs can be improved over that of flat relational DBs will be formed by:

- Taking advantage of opportunities in properly structuring the relations in the first place (i.e., NR DB design, or NR "normalization").
- Proper use of adapted implementations of relational operators which use these structures to their benefit (the lion's share of developing a NR query optimizer).
- Sound decisions on what type of storage structures to use for a particular DBMS (on the basis of workload and application).

Therefore, we believe that the most beneficial initial work in this area will involve studying the effect of application (type of queries normally issued), workload (distribution of queries occurring in a particular application), and storage structures used on the performance of a NRDB.

As a result, we have defined the following problems to be addressed:

1. Develop a rule-based query optimization capability for Nested Relational DBMSs.
2. Study the effect of workload and storage structure on the performance of a nested relational DBMS.
3. After the analysis performed in 1 and 2, a need for non-existing operators, access methods, etc may be discovered. Investigate such needs as they are discovered and provide solutions.

Finally, our research goals include:

1. Develop a set of algebraic rules for performing transformations of nested relational operators within a query tree.
2. For each of the three basic storage models (FSM, DSM, NSM) defined by [HO88b], develop cost functions which characterize the cost of applying available nested relational operators.

3. Using the techniques and theory developed by Freytag [Fre87] and Graefe [GD87, Gra87], develop a rule-based query optimizer which will generate execution plans based on the algebraic transformations and cost functions developed above.

   This will result in a method of generating (predicting) cost information (in terms of CPU time and the number of disk accesses required) associated with particular queries (expressed in terms of nested relational algebra) intended to be executed within a nested relational DBMS based on a known storage methodology.

4. Develop realistic workload information (a "representative" set of queries and distributions of these queries) for a particular application area (e.g., VLSI CAD or Structured Analysis (SA) Design).

5. Design and carry out an experiment based on *predicted* cost information derived from the query optimizer which determines the effect of workload and storage structure on the performance of a NRDB.

6. Validate predicted results via actual implementation (through the use of EXODUS, with the support of MS students).

7. Expand the scope of both predictive and actual performance studies by adding a second application area and corresponding workload study.

8. Add the capability to process SQL/NF [RKB87] statements rather than nested relational algebra expressions.

3 Nested Relations

When one assumes that the value of an attribute must be "atomic," the resulting database is said to be in *First Normal Form* (1NF). The concept of a *nested* relation involves the relaxation of this "atomic attributes only" requirement. As a result, nested relations can produce a much more compact and intuitive representation of the relationships which the database is intended to record.

In 1982, Jaeschke and Scheck [JmS82] presented research involving the creation of unnormalized (nested) relations from normalized (1NF, or flat) relations and what operations should be defined for nested relations. In [FT83], Fischer and Thomas expand the work of [JmS82] by generalizing the nest and unnest operators to allow for multiple attributes and multiple levels; extending the definitions of the relational algebra operators in view of the enlarged "scope" of nested relations; and providing results concerning properties associated with the interaction of nest, unnest, and the relational algebra operators.

In the nested algebras discussed above, manipulations of deeply nested relations will involve first unnesting to the point where the desired "sub-relation" is "exposed" and renesting when the operation has been carried out. This costly, unnecessary, and sometimes destructive process is alleviated by the introduction of *recursive* algebras. These algebras provide "direct access" to required sub-relations and are discussed by [Jae84, SS86, DL87, Col89].

4 Query Optimization of Nested Relations

In order to study the potential performance of the nested relational data model, one has to be able to associate potential costs with a particular query. This is essentially the job of the query optimizer (plus the selection of a minimal cost execution plan from within the set of such potential costs).

There has been very little research performed in this area. To our knowledge, research involving the optimization of queries based on the concepts of nested relations and complex objects has been limited to

1. The algebraic optimization of queries expressed in *flat* relational algebra based on a knowledge of *nested* physical storage structures [Sch86],

2. The algebraic optimization of recursive nested relational algebra expressions [Col89],

3. The development of access plans and the analysis of access costs for Summary Table By Example (STBE) queries [OMO85] (essentially non-recursive nested algebra expressions proposed by Jaeschke and Scheck).

4. A proposal for the extension of System R's optimizer for handling "complex objects" [LDE+85], and

5. The master's thesis of Bartels and Moeller [BM85]. We have only recently come across these references. This work was evidently intended to be used by the AIM project, however, recent publications by the AIM "team," while mentioning the need for query optimization, do not reference this work.

Developing a functioning query optimizer for nested relations using the EXODUS "optimizer generator" approach will be a primary contribution of our research.

5 Storage Structures

The "heart and soul" of any DBMS has to be the methods by which data is stored, accessed, and manipulated. The basis of these overall data management methods is the way in which data is physically stored. For any "real-life" system, this, in turn, means how the data is structured on secondary (disk) storage devices.

Unlike query optimization, the subject of storage structures for nested relations *has* been fairly well explored. This research has arisen from efforts to develop a DBMS based on the concept of nested relations [PSS+87, Dad88, LKD+88, KCB88, DVG88], as well as individual investigations into storage structures which will best support the use of nested relations [HO88a, HO88b, DPS86, VKC86]. In reviewing these results, one finds that a common set of requirements that any acceptable storage structure must have would include:

- The ability to "quickly" access either an entire nested relation or any sub-relation within the overall structure of a nested relation.

- A nested relation should be capable of growing or shrinking "in place" as tuples within sub-relations are added or deleted, without having to constantly re-organize the entire nested relation.

- Any implementation must allow for variable-length attributes (atomic or set-valued).

- Sub-relations should be clustered as closely as possible to the relation under which they are nested. The idea here is that an access to nested relations which follows the hierarchical schema should be able to obtain the necessary pages from disk in sequential order.

- Traditional access methods such as B+-trees should be supported in order to allow for rapid access to required tuples as is true in existing flat relational DBMSs.

In [HO88b], Hafez and Ozsoyoglu present a method of classifying the storage structures which have been proposed for the support of nested relations. In this classification scheme, there are three basic storage models (Decomposed, Normalized, and Flattened) and three further "hybrids" (Partial Decomposed (P-DSM), Partial Normalized (P-NSM), and Partial Flattened (P-FSM)) which describe those storage models involving elements of two other models. The P-NSM model in particular is discussed in detail in [HO88a]. This paper also describes a proposed algorithm for determining, on the basis of expected queries (workload), the most efficient compromise between a normalized and completely flattened model.

References

[Bid87] Nicole Bidoit. The VERSO algebra or how to answer queries with fewer joins. *Journal of Computer and System Sciences*, 35(3):321–364, December 1987.

[BM85] R. Bartels and J. Moeller. Entwurf und implementierung einer regelbasierenden planungskomponente fuer die optimierung von datenbankanfragen in einer sequel-artigen sprache (design and implementation of a rule-based planning component for the optimization of sequel-like database queries). Master's thesis, Tech. University of Darmstadt and IBM Heidelberg Scientific Center, November 1985. In German.

[Col89] Latha S. Colby. A recursive algebra and query optimization for nested relations. To appear in SIGMOD 89, 1989.

[Dad88] Peter Dadam. Advanced information management (AIM): Research in extended nested relations. *Data Engineering*, 11(3):4–14, September 1988.

[DL87] V. Deshpande and Per-Ake Larson. An algebra for nested relations. Rearch Report CS-87-65, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, December 1987.

[DPS86] U. Deppisch, H.-B. Paul, and H.-J Scheck. A storage system for complex objects. In *International Workshop on Object Oriented Database Systems*, pages 183–195, 1986.

[DVG88] Anand Deshpande and Dirk Van Gucht. An implementation for nested relational databases. In *Proceedings of the Fourteenth International Conference on Very Large Databases, Los Angeles*, pages 76–87, August 1988.

[Fre87] J. C. Freytag. A rule-based view of query optimization. In *Proceedings of ACM-SIGMOD 1987 Annual Conference, San Francisco*, pages 173–180, 1987.

[FT83] Patrick C. Fischer and Stan Thomas. Operators for non-first-normal-form relations. In *Proceedings of the 7th International Computer Software Applications Conference, Chicago*, pages 464–475, November 1983.

[GD87] G. Graefe and D. DeWitt. The exodus optimizer generator. In *Proceedings of ACM-SIGMOD 1987 Annual Conference, San Francisco*, pages 160–172, 1987.

[Gra87] Goetz Graefe. *Rule-Based Query Optimization in Extensible Database Systems.* PhD thesis, University of Wisconsin-Madison, 1987.

[HO88a] Aladdin Hafez and Gultekin Ozsoyoglu. The partial normalized storage model of nested relations. In *Proceedings of the Fourteenth International Conference on Very Large Databases, Los Angeles*, pages 100–111, August 1988.

[HO88b] Aladdin Hafez and Gultekin Ozsoyoglu. Storage structures for nested relations. *Data Engineering*, 11(3):31–38, September 1988.

[Jae84] Gerhard Jaeschke. Recursive algebra for relations with relation valued attributes. Technical Report 84.01.003, Heidelberg Scientific Center, IBM Germany, 1984.

[JmS82] Gerhard Jaeschke and Hans Jörg Schek. Remarks on the algebra of non first normal form relations. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles*, pages 124–138, March 1982.

[KCB88] W. Kim, H. Chou, and J. Banerjee. Operations and implementation of complex objects. *IEEE Transactions on Software Engineering*, 14(7), July 1988.

[LDE+85] V. Lum, P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill. Design of an integrated DBMS to support advanced applications. In *Proceedings of the International Conference on Foundations of Data Organization, Kyoto*, pages 21–31, May 1985.

[LKD+88] V. Linnemann, K. Kuspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Sudkamp, G. Walch, and M. Wallrath. Design and implementation of an extensible database management system supporting user defined data types and functions. In *Proceedings of the Fourteenth International Conference on Very Large Databases, Los Angeles*, pages 294–305, 1988.

[OMO85] Gültekin Ozsoyoğlu, Victor Matos, and Z. Meral Ozsoyoğlu. Query processing techniques in the summary-table-by-example database query language. Technical report, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH, 1985.

[PSS+87] H.-B. Paul, H.-J. Schek, M.H. Scholl, G. Weikum, and U. Deppisch. Architecture and implementation of the darmstadt database kernel system. In *Proceedings of ACM-SIGMOD 1987 Annual Conference, San Francisco*, pages 196–207, 1987.

[RKB87] Mark A. Roth, Henry F. Korth, and Don S. Batory. SQL/NF: A query language for ¬1NF relational databases. *Information Systems*, 12(1):99–114, 1987.

[Sch86] Marc H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *International Conference on Database Theory, Rome (Lecture Notes in Computer Science 243)*, pages 380–396. Springer-Verlag, 1986.

[SS86] Hans-Jörg Schek and Marc H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[VKC86] Patrick Valduriez, Setrag Khoshafian, and George Copeland. Implementation techniques of complex objects. In *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto*, pages 101–110, August 1986.

An Operational Optimization Approach for Parallel n–join with Large Number of Processors

Li Tong

Nick Roussopoulos[†]

Department of Computer Science
University of Maryland
College Park, MD 20742

March 1989

The problem we are investigating is the optimal completion time of parallel join of n base relations (n–join). This problem has been well recognized and previous research work for sequential n–join can be seen in [1][2][3]. We found that the problem is entirely new for parallel case and new operational approach has been proposed. Parallel Processing Tree (PPT) is used to model parallel non–pipelined n–join [4]. We proved that obtaining an optimal PPT in general is NP Complete even for the tree queries. A set of transformation operations of the PPT were defined and an algorithm for building up a canonical PPT for any query by applying these operations has been presented. A canonical PPT has the nice properties that are required to perform a set of joins efficiently. Among them are high parallel degree, balanced dataflows, good selection of source and target relations. Another advantage of these operations is that they can also be applied to adjust the canonical PPT dynamically during the execution of the joins if some unexpected delay is encountered in some of the processes.

We feel that PPT is a very convenient and powerful model for capturing both sequential and parallel behavior of n–join. It can model both pipelined and non–pipelined parallel joins. The dynamic aspects of the PPT allow adjustments of the processes during the join execution. This can be used for controlling the number of joins executed concurrently in an environment with limited number of processors.

The results in this paper have general purpose. They do not restrict to any two way parallel join method, any parallel join algorithm which breaks down to a set of sequential joins running on single processors respectively and the sequential join has the cost model satisfies the formula proposed by will be in the optimization scope of this paper. Neither

[†] tong@mimsy.umd.edu, nick@mimsy.umd.edu

do the results rely on any specific DB machine architecture. They are applicable to any multiple processor system in which several transactions are running parallelly. The processors can be either tightly coupled or loosely coupled as long as each secondary storage unit can be accessed by all the processors and accessing to different storage unit by different processors can be done parallelly. Following are the illustration of the PPT operations that are defined to transform an arbitrary PPT into a canonical one. Due to the limitation of the space, we include here only the description directly relate to the operations, all other details are not included.

The first operation is *flipping*. Consider the component given in Fig.1 (a). The flipping operation transforms it into the component shown by Fig.1 (b) if $G(rho_{xy}, y) < G(rho_{yx}, x)$, otherwise the result is shown by Fig.1 (c). Notice that $\rho_{xy} = \rho_{yx}$. In Fig.1 (a), y must be the root of the leftmost children component of x. C_1 and C_2 in Fig.1 (a) (b) and (c) are the sets of children components of y and x respectively in Fig.1 (a). Similar representation is used for the other figures in this section.

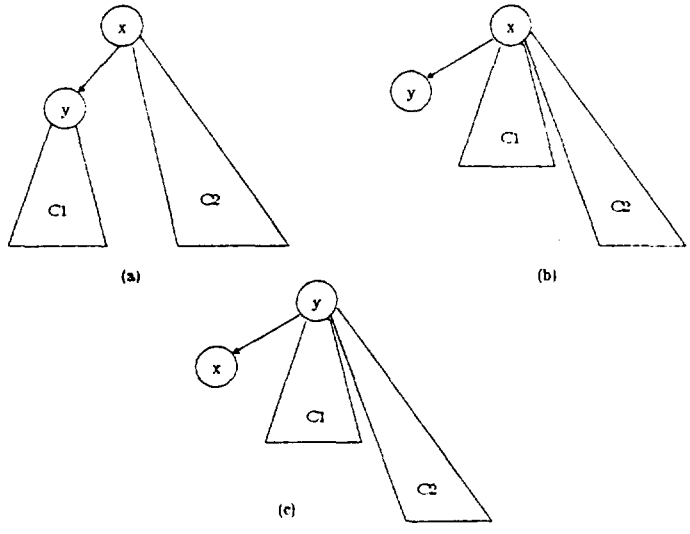


Fig. 1 Illustration of Flipping

The intuition and the effect of flipping is to parallelize a number of otherwise sequential joins. For example, the join of base relations x and y is paralleled with other joins in C_1 and C_2 after the flipping. Initially, the join represented by the edge from x to y in Fig.1 (a) is activated after the completion of the component rooted at y.

The second operation is called *rotation*. It balances the two designated relations to an edge in case one relation has been produced much earlier than the other, and thus, has to wait. Given a component represented by Fig.2 (a) where all the children components of any root are in the optimal order, the rotation transforms the component into the one in Fig.2(b) if $G(\rho, C_1') \leq G(\rho, C_2')$. Otherwise the new component is represented by Fig.2(c). Here C_1' is the component rooted at x and containing the children components in C_1, and C_2' is the component rooted at y and containing the children components C_{21} up to C_{2l}, ρ is the join selectivity between C_1' and C_2'. We require that $F(C_{21}) \leq F(C_1') \leq F(C_{2k})$ for k

= i+1, ..., d. According to the ordering of the children components, only left rotation is necessary.

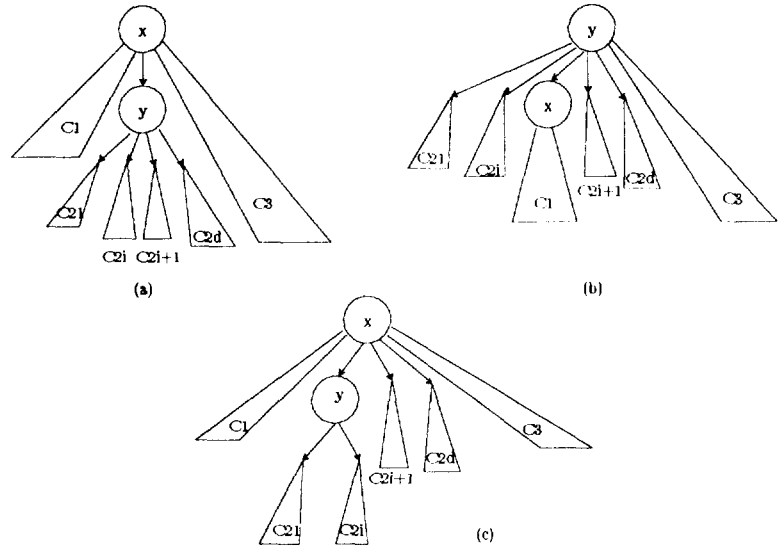


Fig. 2 Illustration of Rotation

For the two relations designated to an edge, the one represented by the component pointed to by the edge is assumed to be the target relation. The source and target relations are well defined by the PPT and can be decided based on the function G value of these two relations. The one with smaller G value is preferred to be the target relation according to the cost formula we have developed for the PPT. The operation *switching* is designed to change the roles of the source and the target relations designated to an edge in a PPT. Given a component defined in Fig.3 (a), the switching operation transforms it into the component shown in Fig.3 (b). The source and the target relations designated to the edge (x,y) have switched their roles.

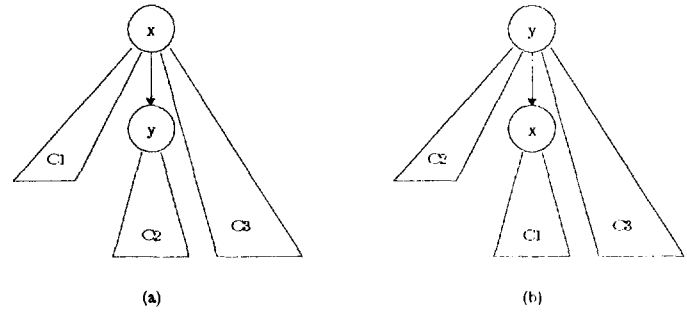


Fig. 3 Illustration of Switching

In our canonical PPT generation algorithm, each application of the operations is always profitable. It turns out that the triggering condition for a profitable application of the operations is very simple. This assets the applicability of our algorithm.

Currently the canonical PPT generation algorithm has been implemented and statistic data are being collected. The preliminary results are very encouraging. For example, with 20 relations, the ratio of the optimal PPT cost to the canonical PPT cost reaches on average 81% with the same amount of processors.

References

[1] T. Ibaraki, and T. Kameda. "Optimal nesting for computing N-relational joins." *ACM TODS* Vol.9, No.3, Sept. 1984

[2] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Optimization of nonrecursive queries." *Proceedings of VLDB 86, Kyoto*

[3] A. Swami, and A. Gupta, "Optimization of large join queries." *SIGMOD RECORD* Vol. 17, No. 3, Sept. 1988

[4] N. Roussopoulos, L. Tong, "Optimization of Response Time in Parallel Pipelined n-Joins." *UMIACS-TR-89-6 CS-TR-2180 Dept. of Computer Science, Univ. of Maryland, College Park, Jan. 1989*

Evaluation of Linear Recursive Predicates in Deductive Databases

Louiqa Raschid
Institute for Advanced Computer Studies
Department of Information Systems
University of Maryland
3129 A.V. Williams
College Park, MD 20742
louiqa@secd.cs.umd.edu

Extended Abstract

We present our research on efficiently evaluating linear recursive predicates in deductive databases. Results of this research have been presented in [RAS86 and RAS89]. Deductive databases integrate techniques from logic programming and database management systems. Recent research has focused on *compiling* logic programs in deductive databases. Compiling a clause of a logic program generates a proof tree, for each intensional database predicate, comprising only extensional database predicates. This compiled clause is equivalent to a relational algebra expression that can be optimized and evaluated efficiently by the relational database management system.

In the presence of recursive predicates, infinite proof trees are generated. To avoid this, a *connection graph* is used to represent the clauses of the logic program; recursive intensions occur in the connection graph as a *potential recursive loop* (PRL) [HEN84]. Traversing the PRL generates a sequence of resolvents or a set of equivalent relational algebra expressions.

The research presented here is a strategy for efficiently evaluating the resolvents generated by a linear recursive predicate. Our approach is to exploit various database query optimization techniques. We identify three techniques, namely query decomposition, sharing of intermediate results and the pipelined execution of relational algebra operations, that are critical to our evaluation.

To describe our optimization, we first decompose each resolvent into a hierarchy of primitive algebraic operations that can benefit from pipelining. We then identify possible parallelism in executing these primitive operations as well as opportunities for intermediate result sharing among the resolvents. Finally, we determine a termination condition to halt the evaluation of the query. Although it is advantageous to maximize the number of primitive operations being executed in parallel, the degree of parallelism is limited by the availability of processors in the evaluation environment. Similarly, the amount of intermediate result sharing among resolvents is limited by the bandwidth and the structure of the interconnection network. As the resolvents become longer (by the repeated traversal of the PRLs), there is increasing opportunity for parallelism and there are several ways to decompose each resolvent into primitive operations. However, to maximize the opportunity for result sharing, it is desirable to decompose each resolvent so that it can share the greatest common sub-expression from a previously evaluated resolvent. To avoid irregularity in the interconnection network and to simplify the network structure, it is desirable to limit the sharing of results only between adjacent resolvents.

An analytical performance evaluation that compared a *distributed* system that supports *horizontal* concurrency of operations with a *pipelined* system that supports both *horizontal* and *vertical* concurrency of operations was used to study the benefits of the query optimization. In a distributed system with horizontal concurrency, operations that are independent of each other are executed in parallel. In a pipelined system with vertical concurrency, as soon as an operation computes a block of data of a result relation, it passes this block to (all) other operations that use this result relation as input. Thus, several dependent operations along the pipeline execute concurrently.

Our evaluation accurately models the pipelined execution of accumulation type operations, i.e., operations such as the relational join where the output rate varies as more input data accumulates. To explain, in a data-

flow based algorithm for an accumulation type operation, the rate of output blocks produced by the operation is determined by the availability of input, as long as the i-th block of input can be completely processed before the (i+1)-th input block is available; i.e., the output rate is determined by the input rate (which is the output rate of the previous operation providing the input). At some point, the input blocks are available at a faster rate than they can be consumed. This is the *critical point* for this operation, and after this point, the output rate is determined by the processing rate of the operation, itself. Our model reflected this behaviour of the pipelined join operation.

The two measures used in the evaluation study are the *response time* (the time to produce the first block of data) and the *execution time* (the time to complete execution), for each resolvent. Expressions for both the response time and the execution time for an operation at a level, m, are defined recursively with respect to operations at levels (m-1) and lower, that provide input along the *critical path* of the pipeline.

We study the performance of this evaluation strategy with and without pipelining, with respect to four parameters. The first parameter is the *block size, B*. The second parameter is the *join selectivity, j*, of the critical path operations that provide input. The join selectivity, j, is not an absolute value but is normalized with respect to the size of the input relations, and is defined as a growth factor. A growth factor of 1.0 implies that the relations sizes along the pipeline remains steady while a value greater than 1.0 implies that the relation sizes grow (increase) along the pipeline and that later operations of the pipeline execute for longer periods. The third parameter is the *number of tuples, N*, of the extensional database relations, against which the resolvent are evaluated. The fourth parameter that we study is the *complexity* of the relational algebraic expressions operations executed at each node in the pipeline.

The performance evaluation proves the benefit of exploiting these database query optimization techniques when evaluating linear recursive predicates in deductive databases. Currently, this evaluation is being extended to the case where there are are multiple potential recursive loops for any linear recursive intensional predicate.

References

HEN84

Henschen, L.J. and Naqvi, S.A., "On Compiling Queries in Recursive First Order Databases," *Journal ACM*, 31,1, 1984.

RAS86

Raschid, L. and Su, S.Y.W., "A Parallel Processing Strategy for Evaluating Recursive Queries," *Proceedings of the International Conference on Very Large Databases*, Kyoto, Japan, 1986.

RAS89

Raschid, L. and Su, S.Y.W., "An Efficient Strategy for Evaluating Linear Recursive Predicates," submitted to *ACM Transactions on Database Systems. Also appears as University of* Maryland Institute for Advanced Computer Studies technical report UMIACS-TR-89-18, 1989.

Query Compilation and Rule Storage in a Knowledge Base Management System

Louiqa Raschid
Institute for Advanced Computer Studies
Department of Information Systems
3129 A.V. Williams
University of Maryland
College Park, MD 20742
louiqa@secd.cs.umd.edu

Extended Abstract

A knowledge base comprises an intensional database of rules that capture problem-solving knowledge and an extensional database of facts. Executing a transaction against a knowledge base is significantly different from executing a DBMS transaction. If we consider a DBMS transaction to comprise a sequence or tree of database operations, then, these operations are compiled, (optimized) and executed; when the transaction commits, changes are made to the database. In contrast, a KBMS transaction cannot be executed against the database, independent of the rule base. For example, retrieval operations against the database are affected by deductive rules that provide new data for retrieval. Updates to the database are affected by constraints from the rule base which must be satisfied before the updates can be made to the database. Updates may also require further operations on the database to maintain consistency.

In many AI environments rules are stored and processed interpretively, in main memory. This proves inefficient if the rule base is very large and cannot be stored in main memory. If the rules must access a database which is disk resident, then the overhead of accessing the disk, each time data is required is excessive. In addition, interactive accesses to the database cannot be optimized, if the rules are being interpreted.

In order to overcome these limitations, we propose the following techniques:

(1) The use of compilation that allows operations in the transaction to be compiled against the intensional rule base. The compiled transaction can be optimized before execution against the database.

(2) The use of hashing to physically cluster large intensional rules bases on disk, so they can be accessed efficiently through a hash table index.

We have studied the use of these techniques in a knowledge base management system based on an object-oriented semantic data model OSAM* [RAS88, SU88]. One of the components of an object class is the rule-based knowledge defined for that object class. These rules must be applied when applying operations against the instances of the object class (both retrieval and updates of the object instances). We briefly describe our research on using these techniques, in this environment.

Compiling Operations of a Transaction

The operations in the transaction that are to be executed against the knowledge base must first be *compiled* against the rule base. In [REI78], compilation is defined as follows: In the presence of an intensional database a query is compiled if it passes through two distinct phases. In the compilation phase, the query is first processed against the intensional database to produce some form of object code. In the second execution phase, the compiled object code is executed against the extensional database alone. If these two phases are not distinct, then the approach is interpretive.

When the intensional rule base is very large, then, compiling the query may prove to be very expensive; rules that will never have their antecedents satisfied will be expanded completely and compiled and they will generate large query trees that require many unnecessary accesses to the database. Interpretation, however, is

more selective and expands only those rules whose antecedents may be satisifed by verification against the database. It is a depth-first strategy which verifies rules one at at time.

We propose a [Match/Modify/Look-Ahead]/Execute method, or [M/M/LA]/E method, to execute a transaction against the knowledge base; this method is a compilation technique but it has the advantage that it only compiles rules that are relevant and whose antecedents must be verified.

Processing in this method alternates between the M/M/LA and the E phases. In the M/M/LA phases, operations in the transaction will be compiled against the intensional rule base. This will result in modifying the transaction and incorporating new operations and rules into the transaction. Only those rules that are relevant and whose antecedents must be verified will be accessed and expanded. The compiled transaction is now executed against the extensional database. In the E phase, the consequents of those rules whose antecedents are satisfied will be applied. If this results in further modifications to the transaction, then, we re-invoke the [M/M/LA]/E phases for the modified portions of the transaction.

The benefits are that the compiled portions of the transaction may be optimized before execution. By alternating between the M/M/LA phase and the E phase, we are able to selectively verify the antecedents of relevant rules and apply their consequents, which further modify the transaction. Thus, we derive some of the benefits of the interpretive approach.

The compilation assumes that there are two kinds of rules, differentiated on the basis of the antecedents, namely, value independent rules (VIRs) and value dependent rules (VDRs). The antecedents of VIRs test the execution of operations against object classes or the execution of VDRs defined for object classes. The antecedents of VDRs are well-formed formulae describing a relationship among object instances that must be verified against the database. When the antecedents of VDRs are verified, they generate retrieval operations against object classes, that must be executed. The consequents of VIRs and VDRs can be (a) update operations against object classes, (b) well-formed formulae which are descriptions of derived data, i.e., derived instances of object classes, and (c) VDRs defined for object classes; these result in explicit execution chains of VDRs.

In the M/M phase, operations and VDRs defined for object classes, in the transaction, are matched against VIRs defined for those object classes; if there is a match, then, the consequents of the matching VIRs modify the transaction by appending operations or VDRs. The M/M phase is cycled through repeatedly until it terminates; cycles must be eliminated. In the LA phase, the VDRs in the transaction are expanded; the retrieval operations generated by their antecedents are attached to the transaction. These retrieval operations must also be processed by the M/M/LA phase. The retrieval operations (from the antecedents of the VDRs) are appended at a higher execution level, in order to distinguish them from the VDR at a lower execution level. Cycles that cross execution levels must be identified; if they are caused by (linear) recursive VDRs, then, they can be further optimized. If they do not include VDRs they must be eliminated. The M/M/LA phase terminates when all operations and VDRs have been matched against VIRs.

In the E phase, operations in the transaction are executed against the instances of the object classes in the database. Operations at a lower execution level have precedence, in execution. Next, VDRs at that execution level are executed. Each VDR points to a structure at a higher execution level. The structure comprises a conjunction of retrieval operations representing the antecedent of the VDR and possibly, other operations and/or VDRs that were attached (in the M/M/LA phase for each retrieval operation). The retrieval operations are executed against the instances of the object class and the VDRs are expanded as described. Those VDRs whose antecedents are satisfied during the E phase are marked so that they can be processed further; i.e., their consequents will be applied to the transaction and the M/M/LA/E phase is re-invoked for the modified portions of the transaction.

Currently, we have implemented many of these algorithms. We propose to run some experiments on the BBN Butterfly parallel processor machine.

Efficient Storage of Large Rule Bases

During compilation, we wish to provide efficient access to the disk resident rule base defined for the object classes. The M/M phase accesses VIRs whereas VDRs are accessed during the LA phase.

VIRs are defined for a single object class; they are clustered on the disk by the unique name of this object class. Since VIRs are selected, in the M/M phase, using the object class and the operation type or rule name

specified in the antecedent, this clustering scheme is adequate. VDRs must have a unique rule name within the unique object class for which they are defined. Their antecedents will test instances of several object classes; at least one of those object classes must be the one for which it is defined. VDRs are physically clustered on disk in two different ways. If they are explicitly selected for execution as a consequent of a single rule, (either a VDR or VIR), then they are physically clustered with that rule. If they are explicitly selected for execution by several rules then they are physically clustered by the unique object class for which they are defined.

A hash table is maintained in main memory. It is an array ObjTbl. Each element of the array points to a linked list of elements ObjTblE. Each element ObjTblE is a structure comprising the name of an object class, two file pointers (or block pointers) to the VIRs and VDRs defined for that object class, and a pointer to the next element in the list.

The array ObjTbl is first initialized before processing any transactions. For each object class, a hash function is used to obtain an index, i, into the array. Since many object classes may hash to the same location, we traverse the list pointed to by ObjTbl[i] until we find the end of the list. A new element ObjTblE is created for the object class. Next, we determine the physical block (or file) pointers for the VIRs and the VDRs defined for this object class. The values are entered in the ObjTblE element.

During the M/M phase, the ObjTblE structures for the appropriate classes are accessed to obtain the location of relevant VIRs. The consequents of the VIRs are used to modify the transactions. If the consequent of a VIR explicitly executes a VDR, then the current block, (or the subsequent record of the file), is scanned to see if the VDR is clustered with the VIR. If this is the case, then, an additional file (or block) pointer is stored with the transaction element.

During the LA phase, VDRs in the transaction are expanded. If there is already a pointer, then, the VDR is directly accessed. If not, the unique name of the object class for which the VDR is defined is used to access the hash table index (ObjTblE) and thus, the location of the VDR. If the consequent of this VDR explicitly executes another VDR, then, the current block or subsequent file segment is scanned to determine if the VDR is clustered here.

Currently, we are running experiments with the hash table index. We plan to use this index to help identify cycles. Subsequently, we plan to compare other access methods to the hash table index.

References

[REI78]
Reiter, R., "Deductive Question Answering on Relational Data Bases," in *Logic and Databases*, 1978.

[RAS88]
Raschid, L. and Su, S.Y.W., "A Transaction Oriented Mechanism to Control Processing in a Knowledge Base Management System," *Proceedings of the International Conference on Expert Database Systems*, 1988. Also University of Maryland Institute for Advanced Computer Studies technical report UMIACS-TR-89-44.

[SU88]
Su, S.Y.W., et al, "An Object-Oriented Semantic Association Model (OSAM*) for CAD/CAM Databases," *AI in Industrial Engineering and Manufacturing*, AIIE, 1988.

Optimization of Complex-Object Queries in PRIMA - Statement of Problems

Harald Schöning
University Kaiserslautern
P.O.-Box 3049
6750 Kaiserslautern
Federal Republic of Germany

Abstract

The MAD model allows the dynamic construction of complex objects via implicit joins (using an identifier / reference concept). This leads to new questions for optimization. Only the problems are addressed; solutions cannot yet be presented.

1. Basic Features of the MAD Model

One approach to extend the relational model for complex object handling is the concept of identifier (surrogate) and reference. The values of reference attributes consist of values of the identifier attributes of the referenced atoms. Thus, arbitrary directed graphs can be defined via reference values. The strength of such an approach depends highly on the means by which the object structures defined this way are used to describe the object as a unit, including object specification without explicit formulation of joins and selections of objects based on their structure and on their components' values.

The molecule-atom data model (MAD model [1]), which is implemented in the PRIMA project [2] also uses a symmetric identifier / reference concept to support dynamically defined object structures.

Binary relationships among basic objects (called *atoms*, which correspond to tuples in the relational model) are represented by a pair of reference attributes. On the type level, a relationship type between atom types $\mathcal{A}$ and $\mathcal{B}$ is represented by a reference attribute ab in atom type $\mathcal{A}$, and a corresponding reference attribute ba in $\mathcal{B}$. Thus, the atom types are nodes in a graph, where the pairs of reference attributes (i.e. the relationship types) install the edges (*atom type network*). A similar graph can be found on the occurrence level, where the nodes of the graph are atoms (*atom network*). If the reference attribute ab of an atom of type $\mathcal{A}$ with identifier value a contains the identifier value b of atom type $\mathcal{B}$, the reference attribute ba of b must contain a. The traversal of a relationship is supported symmetrically in either direction by the concept of the reference attribute pairs. This is the basis for a central concept of the MAD model: the dynamic object structure definition. A complex object type is defined individually for each *query* by selection of a set of atom types and assignment of a direction to some relationships among them. The complex object type must be a directed, coherent subgraph of the atom type network (*molecule type*), containing one root node (*root atom type*). Each occurrence of the root atom type (*root atom*) induces one occurrence of the molecule type. All atoms that can be transitively accessed from the root atom via the specified directed relationships also belong to the molecule. Thus, the molecule is a directed coherent subgraph of the atom network. If, for example, the molecule type definition is $\mathcal{A}$-$\mathcal{B}$-($\mathcal{C}$,$\mathcal{D}$) (direction left-to-right, i.e. $\mathcal{A}$ is root atom type, is related to $\mathcal{B}$, which in turn is related to $\mathcal{C}$ and $\mathcal{D}$), each atom of type $\mathcal{A}$ constitutes a molecule,

even if it is not actually related to a $\mathcal{B}$ atom , while $\mathcal{B}$ atoms without relationship to an $\mathcal{A}$ atom do not belong to a molecule of this type.

In contrast to the common NF^2-Models, the object's structure cannot be reflected on the secondary storage by clustering, because it may change from query to query, and cannot be predicted at storage time. Thus, retrieval of the complex objects must exploit the values of the appropriate reference attributes by performing an implicit hierarchical join. On the other hand, network-like, recursive, and non-disjoint objects can be easily modeled.

The management of atoms in PRIMA is the task of the *access system*, which can be compared to the RSS of System R. It provides two basic mechanisms for retrieval, both capable of attribute projection:

- The scan facility allows the scanning of an atom type, and the option to restrict the result by a search expression (which must be decidable on a single atom). Additionally, if there are different storage structures (e.g. B-Tree, Grid-File) associated with the corresponding atom type, one may choose one of them for this access.
- Additionally, a direct access via the identifier value is provided. This operation is implemented very efficiently, because it is the most frequent access during the construction of a molecule.

Queries in the MAD model deliver a set of molecules of one type, which may be constructed and restricted by a set of quite complex operations, e.g.

- explicit join (without use of reference attributes) and Cartesian Product of molecules. This operation, however, has not the importance it has in the relational model, because relationships among objects are expressed by the reference concept. For the same reason, it is unlikely that more than two molecule types are joined.
- a structure specific and a value specific selection of molecules by a quantified expression which may address all components of a molecule.
- projection and qualified projection, i.e. a value dependent projection of molecule components. In any case, the resulting molecules must be coherent and contain the root atom.

Furthermore, the MAD model allows network-like and recursive object structures, which are formed from hierarchical ones by specialized operators. The basic operator is *Construction of simple molecules*, which builds up a set of hierarchical molecules (by hierarchical joins), and performs selections that can be evaluated on a single molecule (i.e. no query nesting). It is also able to perform projections on atoms and attributes. All other operators get their input from this operator in a pipelined way. Often, however, *Construction of simple molecules* is the only operator representing a query, since in many cases the complex object facilities are sufficient to model the application's objects without explicit join or other higher operators. Therefore, it is necessary to direct one's attention to the optimization of the execution of *Construction of simple molecules*.

2. Optimization Problems

For the reasons stated above, the following considerations concentrate on the optimization of *Construction of simple molecules*. The construction of a molecule without selection and projection requires only a sequence of hierarchical joins. In contrast to the classical optimization, the join method is fixed for the hierarchical join: Once an atom of a molecule has been found, further atoms of this molecules are fetched by the value of the appropriate reference attributes, i.e. by direct access using their identi-

fier values. Internal mechanisms prevent repeated access to an atom that is addressed by more than one reference.

In order to get all atoms of a molecule, one has to traverse all its relationships in the specified direction, i.e. one has to start with the root atom. The sequence of the subsequent joins is of little importance, since all possible sequences yield the same number of atom accesses and the same result size. If a projection is specified on attributes, it can be passed on to the access system. Since resulting molecule structures must be coherent, projections on atoms can cut away only complete subtrees of the structure. Nevertheless, this is not unlikely to happen, since queries may refer to predefined molecule structures, which may contain more atom types than actually desired. In this case, the projection is carried out by just omitting the corresponding joins.

The problem becomes harder when a restriction is posed on the molecule set to be delivered by *Construction of simple molecules*. It would be inefficient to construct a molecule completely before evaluating the restriction, if only parts of the molecule are referenced in the corresponding expression. Therefore, one can identify two phases of a molecule's construction: the first phase checks the qualification of the molecule with respect to the restriction, the second one completes its components. While the second phase corresponds to the case discussed above, the first phase poses questions concerning execution control of the hierarchical joins. The goal of optimization decisions should be to minimize the average number of atom accesses to determine the *disqualification* of a molecule. This fact shall be illustrated by an example: Consider molecule type $\mathcal{A}$-$\mathcal{B}$-($\mathcal{C}$,$\mathcal{D}$), and the restriction: (**EXISTS** $\mathcal{B}$: $\mathcal{B}.Att2>7$) **AND** (**FOR ALL C**: $\mathcal{C}.Att1=5$) (there must be at least one $\mathcal{B}$ atom with attribute $Att2$ having a value greater than 7, and all $\mathcal{C}$ atoms of the molecule must have the $Att1$ values 5). The first phase will consider atom types $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. $\mathcal{A}$ atoms must be considered to determine the scope of the **FOR_ALL** quantifier. Certainly, the last part of the condition is the strongest, but in order to access $\mathcal{C}$ atoms, one has to retrieve the corresponding $\mathcal{B}$ atoms. In this case, a promising strategy might be depth-first search to check the condition on the $\mathcal{C}$ atoms as early as possible. In other situations, breadth-first or mixed strategies (*execution schedules*) are better, depending on the selectivity of the conditions. In order to estimate this selectivity, one has to assume that the selectivity does not depend on the specific complex object under consideration. This assumption is a bit dubious. In this context the question arises, as to how the selectivity of atom type connecting conditions is to be estimated, e.g. of conditions of the form (**FOR_ALL** $\mathcal{B}$: **EXISTS** $\mathcal{C}$: $\mathcal{C}.Att1=\mathcal{B}.Att2$), which are *not* join conditions in our case.

The problem to find an optimal execution schedule for the hierarchical joins only partially overlaps with the classical join order problem, because the problem cannot be described as a choice of relational algebra terms (such as "choose $\mathcal{A}\times(\mathcal{B}\times\mathcal{C})$ or $(\mathcal{A}\times\mathcal{B})\times\mathcal{C}$?"). Furthermore, representations have to be found to represent the various schedule choices in the query evaluation plan.

The first phase need not always take the root atom type as entry point to the molecule: very selective existential quantified conditions on other components can be evaluated first, followed by a navigation to the appropriate root atom, if a corresponding relationship exists. Cardinality restrictions of the schema often guarantee this existence.

Until now, only storage structure independent decisions were discussed (because the hierarchical join is supported by an efficient mechanism). Of course, the existence of storage structure has to be considered when choosing the entry point for a molecule's construction. If one existential quantified factor of a conjunctional form restriction can be evaluated on single atoms of one type, this restriction may be used to determine the starting point of molecule construction by using it as a restriction of a

scan which uses an appropriate storage structure, if available. In special cases, it may also be useful to compute result sets from different entry points in the same molecule by identifier set manipulations, as the following example suggests: Assume a molecule type $\mathcal{A}$-($\mathcal{B}$,C) with highly selective existential quantified conditions on $\mathcal{B}$ and C supported by an index, and a huge number of $\mathcal{A}$ atoms and related $\mathcal{B}$ and C atoms. In this case, the intersection of the union of all *ba* attribute values of the qualifying $\mathcal{B}$ atoms, and the union of all *ca* attribute values of the qualifying C atoms determine the identifiers of the $\mathcal{A}$ atoms of the qualifying molecules. These considerations of course have to be combined with those about execution schedules.

In order to accelerate access to molecules with a certain structure, the database administrator may explicitly specify clustering criteria (in terms of molecule structure definitions, which are used to *redundantly* store replicas of the data). Access to molecules of this structure should be possible without hierarchical joins, since the access systems supports access to the whole cluster. The question, however, is how such clusters can be used when accessing molecules with a *similar* structure. This decision again depends on the degree of similarity as well as on the restrictions associated with the molecules, and is further discussed in [3].

The PRIMA system is designed to run on a closely-coupled multi-processor machine [4]. Thus, parallelism has to be specified in the query evaluation plans. Concerning Construction of simple molecules, the second phase of molecule construction can be fully parallelized, i.e. all atoms whose identifier values are known, can be accessed in parallel. In the first phase, however, introducing parallelism and saving atom accesses are contrary goals. A compromise between run time reduction by parallelism and overhead by unnecessary atom accesses has to be found, and the strategy descriptions mentioned above have to be enhanced by parallelism specifications. Furthermore, cost formulas have to be developed to combine the costs of parallel executions (cost are not additive any longer). Finally, it must be investigated, whether parallel construction of different molecules [5] is not superior to parallelizing the construction itself (easier to control and to determine), or if both methods should be combined. This also depends of the number of available units of parallel execution (processors), which will range from 2 to 30 in the PRIMA project. Note that parallel methods like hash partitioned join do not help in this environment.

Another goal of the PRIMA project is to support efficiency enhancing actions (such as creation of indices, clustering, etc.) by the *system*, i.e. at least propose the installation or release of certain storage structures, or even undertake these actions itself. To be able to do this task, the system must carry detailed statistic describing access patterns to the data. We do not yet know, which information is to be kept in these statistics and how they will burden the system at query execution time.

PRIMA is designed to be extensible. Thus, the optimizer must be rule-driven in any way, in order to be able to integrate new optimization directives easily. The problem is to find an appropriate language for these rules, since dealing with complex object structures will require complex rule structures. On the other hand, complex rules may require a certain amount of overhead to decide their applicability.

3. Summary

Some optimization problems occurring in the PRIMA project were identified:

- The classical join order and join method problems do not apply to the MAD model; here we have to solve the hierarchical join schedule problem.

- This problem covers the question of entry point selection and search strategies as well as the amount of parallelism to be exploited within each hierarchical join.
- Join sequences can be replaced by cluster accesses. This must be covered by the plan generation within the optimizer. On the other hand, the system should propose clustering structures which are useful for as many queries as possible.
- It is not clear, whether every possible kind of parallelism really should be exploited, when the amount of parallelism supported by the hardware can be reached without doing so.
- The kind information to be kept in the system statistics must be determined.
- A language for the formulation of optimization rules has to be developed.

4. References to Papers Concerning MAD and PRIMA Specific Problems

[1] Mitschang, B.: Towards a unified view of design data and knowledge representation, in: Proc. 2nd Int. Conf. on Expert database Systems, April 1988.

[2] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. 13th VLDB, 1987.

[3] Schöning, H., Sikeler, A.: Cluster Mechanisms Supporting the Dynamic Construction of Complex Objects, in: 3rd Int. Conf. on Foundations of Data Organization and Algorithms, Paris, 1989.

[4] Härder, T., Schöning, H., Sikeler, A.: Evaluation of Hardware Architectures for Parallel Execution of Complex Database Operations, in: Proc. 3rd Annual Parallel Processing Symp., Fullerton, 1989.

[5] Härder, T., Schöning, H., Sikeler, A.: Parallelism in Processing Queries on Complex Objects, in: Proc. Int. Symp. on databases in Parallel and Distributed Systems, Austin, Texas, Dec. 1988.

INDEX SELECTION USING REORDERING TECHNIQUES

Marc Woyna
University of Illinois at Chicago

Dina Bitton
Stanford University and University of Illinois at Chicago

1. Introduction

One of the most difficult aspects of physical database design is the selection of access paths. Since a relational database system utilizes only available predefined access paths, the database designer is responsible for determining the set of access paths which balance efficient data retrieval against storage overhead and maintenance costs involved in updates, deletions, and insertions. A poor choice of primary and secondary access paths in a relational database can result in serious performance problems.

The purpose of this paper is to present a heuristic based algorithm for secondary index selection in relational databases which produces near-optimal solutions with considerably smaller computational cost. The index selection problem concerns finding an optimal set of indexes that minimizes the average cost of processing queries. Cost is measured in terms of the number of I/O accesses required to process the queries. The algorithm presented uses a reordering technique to order the columns in the database according to an estimation of their usefulness for indexing, and a cutoff heuristic to determine which columns to index from the ordered list. In an extensive test to determine the optimality of the algorithm, it found optimal or near-optimal solutions in all cases. The time complexity of the algorithm shows substantial improvement when compared with the approach of exhaustively searching through all possible alternatives. The algorithm is designed to incorporate multiple-file databases for which clustering columns have been previously determined and secondary indexing is limited to single columns.

2. Reordering Techniques

Reordering techniques are useful when one wishes to heuristically reduce a search space while retaining a near optimal solution. Rather than attempt to remove unwanted objects from the search space, an attempt is made to order the search space such that unwanted objects appear near the bottom of the list. Initially, an ordered list of objects is produced using some technique such that the ordering is an estimate of the objects' usefulness, with the

most useful object at the head of the list and the least useful object at the tail of the list, although the technique can be applied to a randomly ordered list. Next, using some criterion, the list is iteratively reordered, with the result that useful objects move forward in the list. Reordering continues until no further reordering is possible or some other condition is reached. The resulting ordered list of objects is considered optimal if objects 1 to n are equivalent to the optimal object set of size n.

The Δ C reordering criterion has been developed and has been proven effective in problems in which a non-heuristic cost for a set of objects can be calculated (Gose and Wu, 1973). The Δ C method for reordering assigns a value to each object, Δ C, calculated as follows. For each object n, Δ C is equal to the difference in cost between the set of objects 1 to n, and the set of objects 1 to n - 1. For example, if adding object 15 to the set of objects 1 to 14 results in a cost that is less than the cost of set 1 to 14, the Δ C for object 15 is negative, reflecting a decrease in cost associated with adding object 15 to the resultant set. Intuitively, the greater the reduction in cost due to the inclusion of an object into the object set, the more likely the object will be a member of the optimal set.

Calculating the Δ C for n objects requires n+1 steps. Initially, the cost of the empty set is calculated. This set is the resultant set. Next, the object at the head of the input list is added to the resultant set, and the cost of the new set is calculated. The difference in cost, Δ C, between the new set and the old set is assigned to the first object. This process continues until all n objects from the input list have been added to the resultant set, and each assigned a Δ C.

The resultant set is then reordered according to this Δ C estimate such that objects having a lower Δ C are moved forward in the list, producing a new ordered list of objects. If the new ordered list differs from the original input list, the process is repeated. It may be necessary to restrict the number of iterations if thrashing occurs, or if the computational costs become excessive. However, in applying the technique to a set of 157 features used for automated pattern recognition of electrocardiograms Gose found that approximately 20 iterations were sufficient to order the set (Gose, 1974).

3. Index Selection Using Reordering Techniques

Applying reordering techniques to index selection involves the selection of a reordering criterion. Since it is possible to calculate the cost of processing a set of queries on a scheme with a given index configuration, the Δ C reordering technique can be applied. An ordered index set is produced using the Δ C criterion such that the index in position n would be chosen which, when included with the set of indexes 1 to n - 1, would reduce the cost of pro-

cessing the query set the maximum. The number of indexes to use is left as a variable to be optimized at a later stage in the design.

The time complexity of the index reordering algorithm is O(k*g*v), where k is the number of iterations performed, g is the number of queries specified in the query set, and v is the number of columns in the relation set. The time complexity is estimated in terms of the number of calls to the query cost evaluator, the costliest operation in the process. The cost evaluator is called v times for each iteration of the reordering technique. In each call, the query cost evaluator must calculate the cost of processing each of the g queries, for a total cost of O(g*v) for each iteration.

3.1. Zero Cutoff Criterion

Having ordered the set of n columns using the index reordering algorithm, a set of $m \leq n$ columns must be chosen which will be indexed. The zero cutoff criterion has been developed as a heuristic means of choosing the subset of columns to index. If the resultant ordered list of indexes produced by the index reordering algorithm is assumed to be optimal, the optimal set will be all indexes that have a negative Δ C after the final iteration of the reordering algorithm. However, since the reordering algorithm is not guaranteed to produce optimal solutions, the criterion is a useful heuristic.

4. Experimental Model

4.1. Assumptions

Only non-clustered indexes are considered in the index search space. It is assumed that the relations' clustering column(s) are defined in the logical design phase. It is assumed that each relation is stored in a secondary storage medium which is divided into fixed-size units called blocks. Furthermore, each relation is mapped into a single file and each file contains only one relation. A B^+-tree index can be created for a column of a relation. The leaf level of the index consists of key-TID pairs for every tuple in that relation and the leaf level blocks are chained to assist in the sequential scan of the index. Finally, only conjunctive predicates consisting of equality predicates are considered (e.g. Column A = 'a').

4.2. Query Model

Four types of queries are considered in the index selection problem: select, update, delete, and insert. Each class is restricted to queries involving only one or two relations. This restriction reduces the complexity of the query cost evaluator.

4.3. Cost Model

The cost of processing a query is measured in the number of I/O accesses necessary to complete the request. In determining this cost, only the cost of accessing and maintaining the database and its indexes are considered. The cost of processing a query is calculated in the query cost evaluator. The query cost evaluator calculates the cost of processing a query against the database scheme having a given index configuration using cost formulas developed in earlier work on transaction-processing cost estimation (Whang, 1985).

4.4. Data Sets

Forty-eight data sets representing varying database statistics and usage are used to validate the index reordering algorithm. Each data set is composed of four parts, a database scheme, a cardinality set, a query set, and a query frequency set.

A single database scheme consisting of 2 relations with 5 columns each was used. This allowed for both single-table and multiple-table (joins) queries. Optimal solutions are determined by exhaustively searching through all 1024 possible indexing combinations (2^{10}).

Four cardinality sets are used. Each set is composed of a cardinality for each relation in the scheme, a blocking factor for each relation in the scheme, a selectivity for each column of each relation, and a blocking factor for the indexes used in the index configuration.

Two query sets were used. Each query set consisted of 30 queries: 15 select, 5 delete, 5 insert, and 5 update queries, including both single-table and multiple-table queries.

Six sets of relative query frequencies were used. Each set consisted of 30 frequencies, mapped to each query in the query set. Frequency values ranged from 0 to 50. Each set attempted to focus on a different aspect of query processing (i.e retrieval, updates).

4.5. Validation of the Algorithm

Two experiments were performed to validate the reordering algorithm. In each experiment, an index configuration was determined for each data set using both an exhaustive search algorithm, and the reordering algorithm. The results of the two methods were then compared. In particular, the deviations of the reordering solutions from the optimal solution for each data set was measured. In each of the experiments, the zero cutoff criteria was applied after the reordering algorithm to determine if a column should be indexed. In the first experiment, the initial ordering of the columns was a sorted list based on the estimated usefulness of each column when indexed individually. In the second experiment, the columns were randomly ordered before applying the reordering algorithm. The purpose of the second experiment was to determine the benefits, if any, of initially ordering the index set. Additionally, a third experiment was designed to test the reordering algorithm on a larger scheme. Two data sets consisting of a 20 column scheme, 2 query sets, 1 frequency set, and 1 cardinality set were used.

5. Results and Analysis

Extensive testing has demonstrated the feasibility of the reordering algorithm. The results of the experiments are presented in tables in the appendix. For each experiment, the error rate for each data set was calculated as:

Error Rate = | Optimal Cost - Zero Cutoff Cost| / Optimal Cost.

In experiment 1, the algorithm found optimal solutions in half of the input data sets, and had an average error rate of less than 0.2 percent for all data sets, when compared to the optimal solution. In the worst case, the error rate was still less than 4 percent.

The results of experiment 2 show the benefit of ordering the index set before applying the reordering algorithm. Whereas ordering the index set resulted in 24 optimal solutions, the randomly ordered set produced no optimal solutions. In addition, the average error rate was significantly higher than the ordered set, and the average number of iterations was slightly higher.

The low number of iterations necessary to reorder the index set in experiment 1 indicate that the initial order is relatively close to optimal. Thus, the assumption that an index is a good candidate for indexing if it reduces the cost of processing the query set when used alone appears to be heuristically sound.

The benefits of index selection using the reordering algorithm for large schemas are demonstrated in experiment 3. Although a 20 column schema by no means represents a large schema, the cost of finding the optimal index set has already become excessive, particularly if the solution was required as part of an interactive database design session. Additionally, the number of iterations compares favorably with the average number of iterations required for experiment 1, although the number of columns in experiment 3 was doubled. While two data sets are insufficient to draw any strong conclusions, the low error rates produced are an indiction that index selection for larger database schemas will benefit from the reordering algorithm.

6. Summary

In this paper we have presented a heuristic algorithm for index selection in relational databases using reordering techniques. Reordering techniques were presented as a computationally efficient way to reduce a search space by ordering the objects in the search space according to some estimation of the object's usefulness, rather than removing objects. The benefit of this approach is that the number of objects to include in the final search space can be optimized at a later stage in the design.

The feasibility of applying reordering techniques to the index selection problem was demonstrated in extensive testing, producing optimal or near-optimal solutions in all cases. The time complexity for the reordering algorithm, O(k*g*v), is less than previous algorithms which could be applied to multiple-file databases, such as the DROP heuristic (Whang, 1984). In addition, the zero cutoff criteria proved to be an effective heuristic for determining the columns to index, given the complete ordered index set produced by the reordering algorithm.

References

Gose, E.E. and Wu, F.B.H.: Some Second Order Techniques for Selecting Subsets of Pattern Recognition Properties. Proceedings of the Sixth Hawaii International Conference on Systems Science, 73-76, 1973.

Gose, E.E.: Selecting Useful Information for Medical Decision Making. Proceedings of the International Conference on Medical Informatics, 1-10, 1986.

Whang, Kyu-Young: Index Selection in Relational Databases. IBM research report, IBM, Yorktown Heights, NY, 1984.

Whang, Kyu-Young: Transaction-processing Costs in Relational Database Systems. IBM research report, IBM, Yorktown Heights, NY, 1985.

APPENDIX

Table I.
ACCURACY AND PERFORMANCE OF INDEX REORDERING ALGORITHM USING SORTED INITIAL ORDER AND ZERO CUTOFF CRITERIA (10 COLUMN)

Card Set	Query Set	Freq Set	Optimal Cost	Zero Cutoff Cost	Passes	% Error
0	0	0	3045090	3045090	3	0.0
0	0	1	12654530	12655820	2	0.00845
0	0	2	10172780	10173640	2	0.00845
0	0	3	8097760	8099300	2	0.01902
0	0	4	7300	7300	2	0.0
0	0	5	55151650	55151650	2	0.0
0	1	0	3045090	3045090	3	0.0
0	1	1	12654530	12655820	2	0.01019
0	1	2	10172780	10173640	2	0.00845
0	1	3	8097760	8099300	2	0.01902
0	1	4	7300	7300	2	0.0
0	1	5	5013950	5013950	2	0.0
1	0	0	2125270	2125270	3	0.0
1	0	1	9434905	9436135	2	0.01304
1	0	2	6492800	6493600	3	0.01232
1	0	3	6258820	6260360	2	0.02461
1	0	4	5950	6150	2	3.36135
1	0	5	5014700	5014700	2	0.0
1	1	0	2125270	2125270	3	0.0
1	1	1	9434905	9436135	2	0.01304
1	1	2	6492800	6493600	3	0.01232
1	1	3	6258820	6260360	2	0.02461
1	1	4	5950	6150	2	3.36135
1	1	5	5014700	5014700	2	0.0
2	0	0	326450	327020	2	0.17461
2	0	1	1339140	1340990	2	0.13815
2	0	2	1095040	1096200	2	0.10593
2	0	3	863660	865920	2	0.26168
2	0	4	5800	5800	2	0.0
2	0	5	520250	520250	2	0.0
2	1	0	326450	327020	2	0.17461
2	1	1	1339140	1340990	2	0.13815
2	1	2	1095040	1096200	2	0.10593
2	1	3	863660	865920	2	0.26168
2	1	4	5800	5800	2	0.0
2	1	5	520250	520250	2	0.0
3	0	0	318010	318010	2	0.0
3	0	1	1294925	1294925	2	0.0
3	0	2	1042860	1042860	2	0.0
3	0	3	864620	865200	2	0.06708
3	0	4	7250	7250	2	0.0
3	0	5	542850	542850	2	0.0
3	1	0	2621370	2621370	2	0.0
3	1	1	1294925	1294925	2	0.0
3	1	2	1042860	1042860	2	0.0
3	1	3	864620	865200	2	0.06708
3	1	4	7250	7250	2	0.0
3	1	5	542850	542850	2	0.0

Table II.

ACCURACY AND PERFORMANCE OF INDEX REORDERING ALGORITHM USING RANDOM INITIAL ORDER AND ZERO CUTOFF CRITERIA (10 COLUMN)

Card Set	Query Set	Freq Set	Optimal Cost	Zero Cutoff Cost	Passes	% Error
0	0	0	3045090	3045530	3	0.01445
0	0	1	12654530	12655860	3	0.01051
0	0	2	10172780	10173680	3	0.00885
0	0	3	8097760	8101040	3	0.04051
0	0	4	7300	7400	3	1.36986
0	0	5	5013950	5018050	2	0.08177
0	1	0	3045090	3045530	3	0.01445
0	1	1	12654530	12655860	3	0.01051
0	1	2	10172780	10173680	3	0.00885
0	1	3	8097760	8101040	3	0.04051
0	1	4	7300	7400	3	1.36986
0	1	5	5013950	5018050	2	0.08177
1	0	0	2125270	2125740	2	0.02211
1	0	1	9434905	9436310	3	0.01489
1	0	2	6492800	6493820	2	0.01571
1	0	3	6258820	6262160	2	0.05336
1	0	4	5950	6500	2	9.24370
1	0	5	5014700	5019000	2	0.08575
1	1	0	2125270	2125740	2	0.02211
1	1	1	9434905	9436310	3	0.01489
1	1	2	6492800	6493820	2	0.01571
1	1	3	6258820	6262160	2	0.05336
1	1	4	5950	6500	2	9.24370
1	1	5	5014700	501900	2	0.08575
2	0	0	326450	327700	3	0.38291
2	0	1	1339140	1343030	3	0.29048
2	0	2	1095040	1097580	3	0.23196
2	0	3	863660	868620	3	0.57430
2	0	4	5800	6050	3	4.31034
2	0	5	520250	526800	2	1.25901
2	1	0	326450	327700	3	0.38291
2	1	1	1339140	1343030	3	0.29048
2	1	2	1095040	1097580	3	0.23196
2	1	3	863660	868620	3	0.57430
2	1	4	5800	6050	3	4.31034
2	1	5	520250	526800	2	1.25901
3	0	0	318010	318590	4	0.18238
3	0	1	1294925	1296660	3	0.13398
3	0	2	1042860	1044080	4	0.11699
3	0	3	864620	867460	3	0.32847
3	0	4	7250	7900	3	8.96551
3	0	5	542850	546300	3	0.63553
3	1	0	318010	318590	4	0.18238
3	1	1	1294925	1296660	4	0.13398
3	1	2	1042860	1044080	4	0.11699
3	1	3	864620	867460	3	0.32847
3	1	4	7250	7900	3	8.96551
3	1	5	542850	546300	3	0.63553

Table III.

ACCURACY AND PERFORMANCE OF INDEX REORDERING ALGORITHM USING SORTED INITIAL ORDER AND ZERO CUTOFF CRITERIA (20 COLUMN)

Card Set	Freq Set	Query Set	Optimal Cost	Zero Cutoff Cost	Passes	% Error
0	0	0	3105480	3105870	3	0.01256
0	0	1	7910	7910	2	0.0