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:* Consecutive operations can be executed in parallel if they do not require complete operands but can work on tuple packets which are iteratively produced. Pseudo-parallel execution of the operations of such a pipe 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. For parallel execution, a pipe is decomposed into parts which are assigned to different processors, each part consisting of one operation or several consecutive operations.

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, memory-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 resource requirements (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).

In order to minimize execution time the degree of parallelism (defined as the average number of processors used during the execution of the query) has to be increased starting from a plan with (near) minimal 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 the performance of a multiprocessor database machine under several load conditions using query execution plans with differing cost and execution time. Related experiments have been performed in Bubba [Smit89], however under a different objective: to determine the impact of node splitting enforced by data distribution on response times.

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 pipelining and adding 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

$$\pi_{X(Y)}\, \sigma_{p_1}\, E_1 \ldots E_k\, \phi_{X_g F}\, \sigma_{p_2}\, E_{k+1} \ldots E_m\, (R_1 \times R_2 \times \ldots R_n).$$

Each E_i is derived from a subquery and essentially an expression of the same form, however with one additional relation parameter $\mathcal{R}$ in the cross product and possibly including difference, division or outer aggregation (see below). $E\ e = E[\mathcal{R}/e]$ denotes the replacement of $\mathcal{R}$ by the algebraic expression e, thus binding the outer references of the subquery.

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 p 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. Furthermore, it may include semijoin conditions ($\exists e\ p$) as 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 subquery expressions E_i, we also use an outer aggregation $e_1 \bowtie\!\!\lhd_F e_2$ which attaches aggregate function values to each tuple t of e_1 and is equivalent to an outer join followed by the standard aggregation $\phi_{X_1,F}$, X_1 being the attributes of e_1. Arithmetic expressions may be used in projections, selection predicates and aggregations.

Transformations performed in the second step include the replacement of an outer aggregate formation by a standard aggregation (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 outer 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 apprach 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 ([Bry89]).

5 Generating Parallel Execution Plans

To generate parallel execution plans we can introduce additional potential parallelism by changing the order of operations, node splitting and pipelining. 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). Pipelining is enabled by decomposing pipes into parts. Hence the only change in the underlying dataflow program is by 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 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, memory 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 estimates 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. Under these assumptions which are usually fulfilled by a real time operating system, runtime scheduling decisions depend on the following information:

(1) *Assignment of operations to processors.* It has to be fixed only for operations that access base relations. All other operations 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 pipelining 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?*

Proposed solution: We want to find execution plans which minimize response time 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 pipe decomposition 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. 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 pipe decompositions 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. Its implementation will replace 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. A possible implementation technique are dynamic query execution plans [GrWa89].

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

[Bry89] F. Bry: Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. Proc. ACM SIGMOD, Portland, June 1989, pp. 193-204

[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. Proc. 6th Int. Workshop on Database Machines, Springer Lecture Notes, Vol. 368, June 1989, pp. 18-33

[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. 11th 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

[GrWa89] G. Graefe, K. Ward: Dynamic Query Evaluation Plans. Proc. ACM SIGMOD, Portland, June 1989, pp. 358-366

[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

[Smit89] M. Smith et. al.: An Experiment on Response Time Scalability in Bubba. Proc. 6th Int. Workshop on Database Machines, Springer Lecture Notes, Vol. 368, June 1989, pp. 34-57

[SwGu88] A. Swami, A. Gupta: Optimization of Large Join Queries. Proc. ACM SIGMOD, Chicago, June 1988, pp. 8-17