UvA-DARE (Digital Academic Repository)

Analysis of a dynamic query optimization technique for multijoin queries

van den Berg, C.A.; Kersten, M.L.

DOI

[10.1016/0164-1212\(94\)90045-0](https://doi.org/10.1016/0164-1212(94)90045-0)

Publication date

1994

Published in

Journal of Systems and Software

[Link to publication](#)

Citation for published version (APA):

van den Berg, C. A., & Kersten, M. L. (1994). Analysis of a dynamic query optimization technique for multijoin queries. *Journal of Systems and Software*, 27(3), 233-243. [https://doi.org/10.1016/0164-1212\(94\)90045-0](https://doi.org/10.1016/0164-1212(94)90045-0)

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

Analysis of a Dynamic Query Optimization Technique for Multijoin Queries

C. A. van den Berg and M. L. Kersten

CWI, Amsterdam, The Netherlands

Query processing in parallel data base systems is commonly based on the assumption that reliable estimates can be made for the sizes of intermediate results and the load distribution in the system. However, these estimates are difficult to make for nonstandard applications. A solution is to use a dynamic query-processing scheme, which is adaptive to the data and load distribution in a shared-nothing architecture. We propose a novel run-time optimization technique, called task elimination, aimed at reducing the total amount of work for queries over large partitioned data bases. The level of reduction obtained and the total processing cost is determined with a probabilistic model. Finally, we introduce the parallel bottom-up query evaluation strategy, which maximizes the effect of the task elimination optimization technique.

1. INTRODUCTION

Exploitation of distributed processing capacity provided by large multiprocessor systems has been used in many data base machine projects to improve the performance of data base management systems (Boral et al., 1990; DeWitt et al., 1990; Kersten et al., 1988). Although each prototype has demonstrated performance improvement over centralized processing, their assumptions block a performance increase by large multiprocessor configurations (> 32 CPUs). Namely, the query optimizer determines a single, presumably optimal, query evaluation plan based on estimates about the data distribution and the sizes of intermediate results. Furthermore, upon allocation of the subqueries to the available processors, the query optimizer or subquery dispatcher assumes that the query runs in isolation. The effect

of concurrent running queries, which may hold an exclusive lock on a relation fragment or increase the load of one of the participating processors, cannot be taken into account. As a consequence, if the anticipated bottlenecks in a pipelined processing scheme turn out to be wrong or the load on the available processors changes during query evaluation, then the query evaluation plan will be far from optimal.

Dynamic query processing alleviates these problems by adjusting query task allocation to processors and adjusting the query plan at run time. In recent years a number of papers have been published on task allocation and dynamic query optimization that indicate some of the problems and gains to be expected.

Lu and Carey (1986) presented a task allocation algorithm aimed at balancing the system load and minimizing the communication cost for a query evaluation plan. It showed that load balancing leads to significant reductions in the average time a query task waits for I/O and CPU resources.

Bodorik and Riordon (1988) and Nguyen (1981) proposed a scheme based on a threshold mechanism. In their scheme, the query plan is corrected when the actual size of a partial result exceeds the estimated size by a certain threshold value.

Graefe and Ward (1989) introduced the notion of dynamic query evaluation plans to solve the problem of producing query plans for parametrized queries. Query execution involves evaluation of a decision procedure for the actual query constants and the data distribution. Thereafter, the components of an access module are dynamically linked to obtain an optimal execution plan.

Murphy (1989) focussed on performance improvement for query execution on shared memory multiprocessors by use of a minimal number of processors and a limited amount of data base buffers. The

Address correspondence to Carel van den Berg, Researcher, Database Research Group, CWI, Kruislaan 413, 1089 SJ Amsterdam, The Netherlands.

method is based on scheduling page reads and page join operations efficiently.

In this article, we extend our work on dynamic query processing (van den Berg et al., 1991) with a probabilistic model of a dynamic query optimization technique. It highlights the expected gains when our method is used. Furthermore, we present a query evaluation method to maximize the gains of a dynamic query optimization technique.

Similar to the approach of Murphy, we consider query evaluation as a scheduling problem. First, the query is transformed into a query program, which solves the query for a portion of the data base at a single processor. Second, the relations involved are partitioned into fragments. Finally, combinations of these fragments form query tasks, which are executed on the available processors by a centralized scheduler. The query scheduler controls the load balancing and performs logical query optimization by use of up-to-date information on query task execution and the availability of fragments. Our dynamic query-processing scheme aims at improved processing of precompiled parametrized queries, which exhibit large potential parallelism or none at all depending on the parameter settings upon query execution.

In earlier work we focussed on load balancing and load control algorithms for this architecture (van den Berg et al., 1991). We observed a rapid increase of performance toward a maximum that depends on the inherent parallelism in the query only. The overhead incurred by use of a centralized scheduler to manage the load control is negligible in our distributed store environment because of the size of the subqueries.

The novelty of this work is its probabilistic study of the effectiveness of a dynamic query optimization technique called task elimination. It highlights the effect of data distribution patterns on the number of tasks that can be removed by the scheduler. The model also exemplifies the range of fragment sizes, i.e., message sizes, for which the technique is a clear winner. Finally, our cost model shows that parallel bottom-up evaluation, i.e., always pairwise joining of the relations in a query, combined with task elimination leads to impressive total time reductions for small fragment sizes.

The remainder of this report is organized as follows. In Section 2, we describe the dynamic query-processing scheme in relation to static query processing. In Section 3 we analyze the task reduction technique for different data distributions for a single join, which is generalized in Section 4 to a k -way join. Section 5 combines the result of sections 3 and

4 with a processing cost model to determine the total processing cost. Section 6 concludes with a summary and suggestions for future research objectives.

2. DYNAMIC QUERY PROCESSING

The availability of relatively cheap and scalable multiprocessor systems with large amounts of main memory made it possible to consider main memory data base machines (Apers et al., 1992; Bergsten et al., 1991; Palmer Leland and Roome, 1988; Eich, 1988). As the data base grows, more processors can be added to match the memory and processing requirements of the data base applications.

Query processing in this area is generally divided into three distinct phases:

1. Query parsing. This expresses the query as a data flow graph in which the nodes represent the basic operations.
2. Query schedule generation. This consists of query optimization and process allocation. In query optimization, the data flow graph is transformed into a more efficient one by use of an algebra over the basic operations, size estimates of intermediate results, and a set of cost functions. Given the "optimal" data flow graph, the basic operations are allocated to processors that take data adjacency and data transport into account.
3. Query schedule execution. Each processor is given its part of the schedule, the communication between the processes is set up, and the query evaluation is started.

In traditional (static) query processing (SQP) schemes, these phases are performed once for a query. Thus the query schedule and allocation of subqueries to processors is fixed during query evaluation. This could lead to a suboptimal execution due to unreliable cost estimates and resource contention. Furthermore, a change in the query parameters requires query recompilation.

If the query is evaluated in steps, then the schedule can be adjusted at run time to adapt to variations in the data and system load distribution. For this purpose, important performance parameters like the sizes of intermediate results and the current processor load are monitored and fed back to a query scheduler. This scheduler can then choose an optimal query schedule and allocation of subqueries. This technique is called dynamic processing (DQP). A DQP architecture consists, therefore, of a query evaluator and a query scheduler (Figure 1). The query scheduler initiates the query execution by

Figure 1. The general DPQ architecture.

passing a preliminary query schedule to the query evaluator. The latter component can consist of a single processor or several processors, which evaluate the query schedules via pipelining and interoperator parallelism. Each of these processors sends feedback information on the query evaluation characteristics to the query scheduler, which uses this information to create a schedule for pending subqueries.

One approach to dividing the query into steps is the threshold mechanism, in which the query tree produced by the parser is divided into stages, which are evaluated stepwise on the data (Bodorik and Riordan, 1988; Nguyen, 1981).

In our approach, the data are partitioned and the query is evaluated against all combinations of the data fragments by sending the data fragments over the network to processors of the query evaluator. This choice is motivated by considering the following:

- The data base is already partitioned to fit main memory and for locality of reference and integrity enforcement.
- The resources at each processor are limited. CPU and memory-intensive query processing is therefore forwarded to other processors to prevent a processor with a frequently used data fragment from becoming a bottleneck.
- A processor pool is available for dynamic replication of the data base to improve the performance and keep partial results. The pool processors can act as a distributed cache for query processing.

The query result is obtained by taking the union of all partial results. For large relations, this ap-

proach can lead to a very large number of subquery evaluations. For example, if a relation R_i is partitioned into n_i fragments, then $\prod_i n_i$ similar subqueries should be evaluated.¹ Given a large multiprocessor platform, these subqueries can, in principle, be evaluated in parallel. However, the speedup will be limited then by the fragment data preparation or their distribution over the processors. Furthermore, sequential evaluation (or limited parallel evaluation) creates an opportunity for dynamic query optimization: it is possible to reduce the amount of work by use of statistics of previous query evaluations and semantic knowledge of the query operations. In the next section, this dynamic query optimization technique is introduced and explained.

Dynamic query optimization (DQO) is defined as the process of modifying the query schedule based on the measurements taken by the query evaluator. One approach is to use standard query optimization techniques to generate a revised query schedule from scratch at query evaluation time. To control the optimization overhead, the threshold technique can be used to trigger the optimization (Bodorik and Riordan, 1988; Nguyen, 1981). Alternatively, a range of different query schedules could already be prepared before the query evaluation. The measured query statistics then determine the next query schedule (Graefe and Ward, 1989).

In this article, however, we present a run-time optimization applicable to DQP architectures called task elimination. This optimization is based on the assumption that the tuples, which partake in the query result, are generally not uniformly distributed over the product space of the relations involved. Instead, they often exhibit some clustering. As a result, the query schedule need not be evaluated for all the fragment combinations. For example, consider the following query:

```

SELECT *
FROM Person P, City C, Factory F
WHERE P.address = C.name
      and C.name = F.location
  
```

Because the number of factories in a city is variable, there will be many (*City, Factory*) pairs that do not contribute to the query result. Consequently, a large number of (*Person, City, Factory*) combinations do not have to be considered either. This

¹Depending on the query, this number can be reduced by choosing a suitable partitioning function. For instance, use of hash-based partitioning for the operands of an equijoin operation reduces the number of subquery evaluations to $\lceil n_1/h \rceil \lceil n_2/h \rceil$, where h is the number of hash buckets.

knowledge can be represented by the following optimization rule:

$$|join(City, Factory)| = 0 \\ \rightarrow |Q(-, City, Factory)| = 0 \quad (1)$$

This rule expresses the fact that if the result of the join between a *City* fragment and a *Factory* fragment is empty, then the query that uses this combination of fragments is empty regardless of the *Person* fragment. The query evaluator should report an empty intermediate result to the scheduler. The query scheduler can then perform logical optimizations with this rule, i.e., not taking tasks that contain this combination of fragments into execution (Figure 2).

Unlike SQP, the execution order of the joins not fixed in our DQO scheme. The choice between $(Person \bowtie City) \bowtie Factory$ and $Person \bowtie (City \bowtie Factory)$ is determined at run time. Thus, apart from the previous rule, the following rule is also provided:

$$|join(Person, City)| = 0 \\ \rightarrow |Q(Person, City, -)| = 0 \quad (2)$$

Whether rule 1 or rule 2 is used depends on the execution order chosen at run time. If both joins are evaluated simultaneously, then the effect of these optimization rules is combined to reduce the amount of work even further. In section 4 we investigate the effect of taking all pairwise joins into account. This evaluation strategy is called parallel bottom-up evaluation.

3. TASK ELIMINATION

In this section we determine the potential savings that can be obtained by task elimination. The effectiveness of this technique is determined by the frac-

tion of empty intermediate results. Because the detection of empty intermediate results depends on the evaluation order of the operations in the query, we have also examined the effect of using a left/right-deep tree and our own parallel bottom-up evaluation technique on the elimination factor. The results of this exercise can be found in Section 4.

The fraction of empty intermediate results, or elimination factor e , strongly depends on the relational operation and the attribute distribution of the participating relations. The expected value of the elimination factor for a binary operation $\otimes$ between two fragmented relations A and B can be expressed in the probability distribution $P(i, j)$ for empty intermediate results and the number of fragments of the relations n_A and n_B . $P(i, j)$ is defined as the probability that the result of $A_i \otimes B_j$ is empty. Thus

$$P(i, j) = \text{Prob}\{|A_i \otimes B_j| = 0\} \quad (3)$$

$$E[e] = \frac{1}{n_A n_B} \sum_{i,j} P(i, j) \quad (4)$$

A parameter of importance is the fragment size, because it strongly influences the elimination factor. On the one hand, we expect the elimination factor to increase as the fragment size decreases because the probability of an empty result increases, but on the other hand, it also increases the number of tasks, which has a decreasing effect on the elimination factor.

Furthermore, the fragment size determines the processing cost of a task and the communication cost for transporting the fragments between processors. Because it is not possible to present a general cost model for relational queries, we have determined the total processing cost for the specific, commonly used, equijoin query.

Figure 2. Dynamic query optimization.

Before we derive the processing cost for a k -way equijoin, we first determine the elimination factor for a single join operation $A \bowtie_{A.a=B.b} B$. Attribute a is a key attribute of relation A and assumes values in the range $[1, \dots, c_A]$. The relation A is range partitioned over its key attribute a into n_a fragments A_i containing p tuples each, so that the key attribute a of fragment A_i ranges over the values $[pi + 1, \dots, p(i + 1)]$. The relation B is also range partitioned on its key attribute. We assume that the distribution of the key attribute of B and its nonkey attribute b are independent. The fragments B_j also contain p tuples. The attribute value b is distributed according to a certain probability distribution function $\pi(b)$.

To determine the elimination factor $E[e]$, we first express the probability distribution $P(i, j)$ in the probability distribution $\pi(b)$. The probability $P_m(i, j) = \text{Prob}\{|A_i \bowtie B_j| > 0\}$ that the attribute b of fragment B_j lies within the range of key attributes $[pi + 1, \dots, p(i + 1)]$ of fragment A_i is independent of the B fragment, thus

$$P_m(i, j) = P_m(i) = \sum_{b=pi+1}^{p(i+1)} \pi(b) \quad (5)$$

Because $P_m(i)$ is the same for all the fragments of B , we find the following expressions for $P(i, j)$ and $E[e]$:

$$P(i, j) = P(i) = (1 - P_m(i))^p \quad (6)$$

$$E[e] = \frac{1}{n_A n_B} \sum_{i,j} P(i, j) = \frac{1}{n_A} \sum_i P(i) \quad (7)$$

In the following paragraphs we have calculated the elimination factor for the situation in which the foreign key attribute $B.b$ follows the uniform, normal, and Zipf distributions. Because the query is an equijoin operation on a key attribute, the query result has the same cardinality as the referencing relation B . Therefore, the elimination factor can be used to compare the optimization technique for different data distributions.

Uniform Distribution

The uniform distribution is used to find the worst-case behavior for the dynamic query optimization. The reason is that the data contributing to the query result are not clustered, which implies a low task elimination factor for moderately sized fragments. The probability distribution function of the uniform distribution is a constant $\pi(x) = 1/c_A$. From this

distribution we can derive the following:

$$P_m(i) = \frac{p}{c_A}$$

$$P(i) = \left(1 - \frac{p}{c_A}\right)^p$$

$$E[e] = \left(1 - \frac{p}{c_A}\right)^p$$

Normal Distribution

The normal distribution is also used by Schneider and DeWitt (1989) in their performance analysis of join algorithms. This attribute distribution is chosen for our analysis because it could occur in scientific data bases for attributes that represent measurement data. The normal distribution $N(\mu, \sigma)$ is defined by

$$\pi(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(x - \mu)^2}{2\sigma^2}$$

Because this is a distribution of a continuous function, we determine the probability $P_m(i)$ as follows:

$$P_m(i) = \int_{pi}^{p(i+1)} \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(x - \mu)^2}{2\sigma^2} dx$$

Zipf Distribution

In actual data bases, the attribute distribution will more likely follow the Zipf distribution (Salza and Terranova, 1989; Kitsuregawa et al., 1989). The Zipf probability distribution function $Z(c)$ for attribute values in the range $[1, \dots, c_A]$ is defined as:

$$\pi(x) = H_c^{-1} x^{-c}$$

$$H_c = \sum_{k=1}^{c_A} k^{-c}$$

The c parameter is called the decay factor of the distribution. For $c = 0$ the distribution is uniform; if $c = 1$, then the distribution equals the classical Zipf distribution. The distribution of personal income follows $Z(0.5)$.

Data Distribution Comparison

Given the normal, Zipf, and uniform probability distribution functions and equations 6 and 7, we have calculated the elimination factor as a function of the fragment size for different distribution parameter settings (Figures 3, and 4). The uniform distribution is included in Figure 5 because it is equal to a Zipf(0) distribution.

Figure 3. Elimination factor (Zipf).

The graphs show that the elimination factor is a monotonically decreasing function of the fragment size. Furthermore, even the worst-case distribution (uniform) has a potential to reduce the number of tasks for fragment sizes $< 2.5\%$ of the relation size. Finally, we find that the elimination factor is sensitive to the parameters of the distribution. As the attribute distribution becomes more clustered, the task elimination technique becomes more effective over a larger range of fragment sizes (cf. $Z(0.5)$ and $Z(1.0)$).

4. MULTIPLE JOIN EVALUATION

In a multiple join operation, the occurrence of an empty partial join result will also result in the removal of tasks. In this section, the total task elimination E_k of a k -way equijoin is determined given the elimination factors e_i of the $(k-1)$ partial joins. First an expression for the elimination factor for the multiple join is formulated, which is then used to calculate the total processing cost for a specific three- and four-way equijoin.

The evaluation order of the join operations has a strong influence on the total elimination factor. We

Figure 4. Elimination factor (normal). $|A|$, cardinality of relation A .

Figure 5. Sequential evaluation.

considered two different evaluation methods: sequential evaluation, which corresponds to the traditional left- and right-deep query tree, and our own method, parallel bottom-up evaluation.

In the following paragraphs, formulas are derived for a general k -way equijoin query. In the analysis, each joined relation R_i is partitioned into n_i fragments. For each method we derive a formula for the number of tasks N_k that are removed by the task elimination technique. The total task elimination factor of the join query is obtained through division by the total number of tasks N_{task} :

$$E_k = \frac{N_k}{N_{\text{task}}} \quad (8)$$

$$N_{\text{task}} = \prod_{i=1}^k n_i \quad (9)$$

Sequential Evaluation

In the sequential evaluation method, the query is either represented by a left- or a right-deep join tree. The intermediate result at each stage of evaluation can be empty (Figure 5).

Thus, the query evaluator sends the query scheduler information that combinations of two, three, or more fragments result in an empty query result. For a combination of two fragments, many tasks can be removed. However, if the combination is more specific, fewer tasks can be removed. For instance, for a four-way join, the event $|R_1 \bowtie S_1| = 0$ results in the removal of $n_3 n_4$ tasks,² whereas the event $|R_1 \bowtie S_1 \bowtie T_1| = 0$ reduces the number of tasks only with n_4 tasks. The number of eliminated tasks for three- and four-way join operations are given by

$$N_3 = e_1 n_1 n_2 (n_3 - 1)$$

$$N_4 = e_1 n_1 n_2 (n_3 n_4 - 1) + (1 - e_1) e_2 n_1 n_2 n_3 (n_4 - 1)$$

²At least one task had to be executed to generate this event.

Generally, in a k -way join, $e_1 n_1 n_2$ tasks result in empty $R_1 \bowtie R_2$ combinations, because of the first join operation. This results in $e_1 N_{\text{task}}$ eliminations. The next operation results in $(1 - e_1) e_2 N_{\text{task}}$ eliminations, caused by $(1 - e_1) e_2 n_1 n_2 n_3$ empty task results. By summing all terms until the $(k - 2)$ th join operation, we find for N_k the number of tasks that are not evaluated:

$$N_k = \sum_{i=1}^{k-2} \left(\prod_{j<i} (1 - e_j) \right) e_i \prod_{l=1}^k n_l - \sum_{i=1}^{k-2} \left\{ \left(\prod_{j<i} (1 - e_j) \right) e_i \prod_{l=1}^{k-1-i} n_l \right\} \quad (10)$$

Parallel Bottom-Up Evaluation

In the parallel bottom-up evaluation method, all possible join combinations are evaluated in parallel, and the results are subsequently combined (Figure 6). The scheduler is informed if the result of a join for any combination of two fragments is empty. If such an event occurs, the scheduler removes the tasks containing this fragment combination. The number of eliminated tasks in a three- or four-way join operation is thus given as follows:

$$N_3 = (e_1 + (1 - e_1) e_2) n_1 n_2 n_3 - \max(n_1 n_2, n_2 n_3)$$

$$N_4 = e_1 n_1 n_2 n_3 n_4 + (1 - e_1) e_2 n_1 n_2 n_3 n_4 + (1 - e_1)(1 - e_2) n_1 n_2 n_3 n_4 - \max(n_1 n_2, n_2 n_3, n_3 n_4)$$

If we generalize this for the k -way equijoin, we find the following expression for N_k , the number of eliminated tasks:

$$N_k = \sum_{i=1}^{k-1} \left(\prod_{j<i} (1 - E_j) \right) E_i \prod_{l=1}^k n_l - \max(n_1 n_2, \dots, n_{k-2} n_{k-1}) \quad (11)$$

Figure 6. Parallel bottom-up evaluation.

Figure 7. E_4 Zipf distribution. par, parallel bottom-up evaluation; seq, sequential evaluation.

Because all join combinations are evaluated, more work is done than is actually required. However, the additional work invested in a single subquery evaluation will result in a higher total elimination factor and thereby reduce the total amount of work.

Comparison of Evaluation Techniques

By use of equations 8 and 9 and the expressions for the number of eliminated tasks 10 and 11, we have calculated the elimination factor for a four-way equijoin for the normal, uniform, and Zipf distributions for both evaluation techniques (Figures 7, 8). These graphs show that, for all distributions, the parallel bottom-up evaluation results in a larger elimination factor than sequential elimination. For instance, for a fragment size of 2%, an improvement of 15% can be observed for a Zipf(1.0) distribution. However, the gain becomes smaller as the fragment size increases.

Calculation of the elimination factor for other multijoin queries shows that the range of fragment sizes for which the task elimination is effective does not depend on the number of joins, but only on the

Figure 8. E_4 Normal distribution. par, parallel bottom-up evaluation; seq, sequential evaluation; $|A|$ = cardinality of relation A .

distribution parameters. However, within this range, the elimination factor increases with the number of joins.

5. MULTIPLE-JOIN PROCESSING COST

The total elimination factor can now be used to calculate the total processing cost for a multiple-join query. In the cost model below, the assumption is made that the tasks are evaluated by a single processor. Therefore, it gives an upper bound on the total query cost. When more processors are used by the query evaluator, tasks can be evaluated in parallel, which results in a lower response time.³ The following simple cost model can therefore be used to measure the effectiveness.

The total query-processing cost C_{query} for this architecture is determined by the number of tasks remaining after task elimination $(1 - E)N_{\text{task}}$ and the task-processing cost C_{task} .

Because all tasks are executed on a single processor, each task execution for a k -way join requires at most k fragment transports $C_{\text{com}}(p)$ to the processor⁴ and a single multijoin execution $C_{\text{join}}(p)$. These latter factors depend on the fragment size p .

$$C_{\text{query}} = (1 - E)N_{\text{task}}C_{\text{task}}$$

$$C_{\text{task}} = kC_{\text{com}}(p) + C_{\text{join}}(p)$$

Fragment transport requires a constant cost for network access and OS overhead C_{access} and a cost linear in the size of the fragment C_{copy} for copying the data from the network to the processor's memory.

For the execution cost of the join operation we only give an upper bound. Each of the $k - 1$ equi-join operations results in at most p tuple combinations. If we assume a hash join algorithm implementation, then we find that the join cost is also linear in the fragment size. In the first phase of the algorithm a hash table is constructed for one of the join operands, and in the second phase this hash table is probed for each join attribute value of the second operand.

$$C_{\text{com}}(p) = pN_{\text{bytes}}C_{\text{copy}} + C_{\text{access}}$$

$$C_{\text{join}}(p) = (k - 1)pC_{\text{hash}}$$

In Table 1, the parameter setting for our target architecture is given. It consists of MicroVax work-

Table 1. The Parameter Setting for MicroVax Systems that Run Amoeba

C_{access}	Operating system overhead	1 msec
C_{copy}	Data transfer rate	1 msec/1 k
C_{hash}	Hash join cost	100 μ sec
N_{bytes}	Tuple size	0.2 k

stations and uses the Amoeba distributed operating system.

Evaluation of the formulas for these two evaluation methods on a four-way equijoin operation results in the total query-processing cost, as shown in Figures 9 and 10. These graphs present the total query-processing cost as a function of the fragment size for the normal(5,000; 1,500) distribution and the Zipf(0.5) distribution. The elimination factors were obtained by use of the formulas given in section 4, and the cardinality of the relations was set to 10,000.

The combination of task elimination and the cost model illustrate the performance gain to be expected from DQP. The top curve in these graphs represents the total processing cost without task elimination. The result of the calculation shows that, within the effective range of the task elimination technique, a reduction of the total query cost can be obtained as long as the fragment size is small enough. Outside the effective range, the total query-processing cost decreases as the fragment size increases. Therefore, for this simple cost model, if enough memory is available for query evaluation, then it is better to choose a large fragment size outside the effective range of task elimination. However, if parallel query execution is considered, large fragment sizes lead to long communication delays at the query processors, which result in greater response time. There, more research has to be done to study the effect of parallel execution on the effectiveness of the task elimination optimization technique.

Figure 9. C_{query} for a four-way join and Zipf(1.0) distribution.

³Addition of processors influences the effectiveness of the dynamic query optimization technique, because it could be that a processor is processing a task that would otherwise be eliminated by a task, which is executed in parallel. However, this effect is negligible because of the small probability of such an event.

⁴If fragments are properly cached by the processor(s), at most one fragment transport is required.

Figure 10. C_{query} for a four-way join and normal (5,000; 1,500) distribution.

Furthermore, calculations on other multijoin queries show that task elimination becomes more effective as the number of joins increases and the attribute distribution becomes more clustered.

6. CONCLUSIONS

In this article, we presented a general framework for DQP and a detailed analysis of a DQO technique called task elimination. A probabilistic model has been used to estimate the potential gains for different data distributions.

This analysis shows that the task elimination technique can lead to significant reductions in the amount of query tasks that have to be processed. The effectiveness of the task elimination technique depends on the attribute distribution and the fragment size. For a real-life distribution like the Zipf distribution, the results are promising. If enough memory is available, the fragment size chosen should be as large as possible. However, when only a limited amount of memory is available, task elimination and a small fragment size will reduce the total processing cost. Finally, when task elimination was used, it was shown that the parallel bottom-up evaluation method increased the effectiveness of the task elimination method.

Currently, we are attempting to generalize the task elimination technique for other relational operations. Furthermore, we are looking at the effects of fragments caching and fragment size on the total processing cost when the query evaluator is parallelized.

REFERENCES

Boral, H., et al., Prototyping Bubba, A Highly Parallel Database System, *IEEE Trans. Knowl. Data Eng.* 2, 4-24 (1990).

- DeWitt, D. J., Ghadeharizadeh, S., Schneider, D. A., Bricker, A., Hsiao, H., and Rasmussen, R., The GAMMA Database Machine Project, *IEEE Trans. Knowl. Data Eng.* 2, 44-51, (1990).
- Kersten, M. L., Apers, P. M. G., Houtsma, M. A. W., van Kuyk, E. J. A., and van de Weg, R. L. W., A distributed, main-memory database machine: Research issues and a preliminary architecture, in *Database Machines and Knowledge Base Machines* (M. Kitsuregawa and H. Tanaka, eds.), 1988, pp. 353-369.
- Lu, H., and Carey, M. J., Load balanced task allocation in locally distributed computer system, in *Proceedings of the 1986 Conference on Parallel Processing*, 1986, pp. 1037-1039.
- Bodorik, P., and Riordon, J. S., A threshold mechanism for distributed query processing, in *Proceedings of the 16th Annual ACM Computer Science Conference*, 1988, pp. 616-625.
- Nguyen, G. T., Distributed query management for a local network, in *Proceedings of the 2nd International Conference on Distributed Computing Systems*, 1981, pp. 188-196.
- Graefe, G., and Ward, K., Dynamic query evaluation plans, in *Proceedings of the 1989 SIGMOD Conference*, 1989, pp. 358-366.
- Murphy, M. C., Effective resource utilization for multiprocessor join execution, in *Proceedings of the Fifteenth VLDB Conference*, 1989, pp. 67-75.
- van den Berg, C. A., Kersten, M. L., and Shair-Ali, S., Dynamic Parallel Query Processing, Technical Report CS-R9112, CWI, Amsterdam, 1991.
- Apers, P. M. G., Kersten, M. L., Wilschut, A. N., Grefen, P. W. P. J., van den Berg, C. A., and Flokstra, J., PRISMA/DB: A Parallel Main-Memory Relational DBMS, *IEEE Trans. Data Knowl. Eng.* 541-554 (1992).
- Bergsten, B., Couprie, M., and Valduriez, P., Prototyping DB3S, a shared-memory parallel database system, in *Proceedings of the Conference on Parallel and Distributed Information Systems*, 1991, pp. 226-235.
- Palmer Leland, M. D., and Roome, W. D., The silicon database machine: Rationale, design, and results, in *Database Machines and Knowledge Base Machines* (M. Kitsuregawa and H. Tanaka, eds.), 1988, pp. 311-324.
- Eich, M. H., Mars: The design of a main memory database machine, in *Database Machines and Knowledge Base Machines* (M. Kitsuregawa and H. Tanaka, eds.), 1988, pp. 332-339.
- Schneider, D. A., and DeWitt, D. J., A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment, in *Proceedings of the 1989 ACM SIGMOD Conference*, 1989, pp. 110-122.
- Salza, S., and Terranova, M., Evaluating the size of queries on relational databases with nonuniform distribution and stochastic dependence, in *Proceedings of the 1989 ACM SIGMOD Conference*, 1989, pp. 8-14.
- Kitsuregawa, M., Nakayama, M., and Tagaki, M., The effect of bucket size tuning in the dynamic hybrid grace hash join method, in *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, 1989, pp. 257-267.