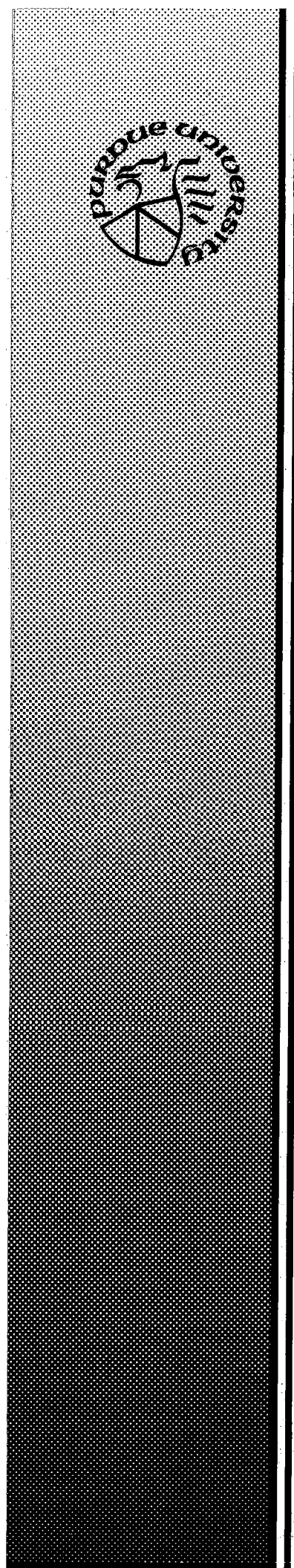
8-1-1989

Algorithm Choice For Multiple-Query Evaluation

Myong H. Kang
Purdue University

Henry G. Dietz
Purdue University



Algorithm Choice For Multiple-Query Evaluation

M. H. Kang
H. G. Dietz

TR-EE 89-50
August, 1989

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Algorithm Choice
For Multiple-Query Evaluation

Myong H. Kang and Henry G. Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
August 1989

ABSTRACT

Traditional query optimization concentrates on the optimization of the execution of each individual query. More recently, it has been observed that by considering a sequence of multiple queries some additional high-level optimizations can be performed. Once these optimizations have been performed, each operation is translated into executable code.

The fundamental insight in this paper is that significant improvements can be gained by careful choice of the algorithm to be used for each operation. This choice is not merely based on efficiency of algorithms for individual operations, but rather on the efficiency of the algorithm choices for the entire multiple-query evaluation. An efficient procedure for automatically optimizing these algorithm choices is given.

1. Introduction

High level relational database query languages such as SQL and QUEL allow users to express what information is desired, not how to obtain it. This enables applications to be independent from details of secondary storage management. The query optimizer is responsible for determining the strategy for efficiently evaluating the queries presented by the user.

Optimization of database queries specified in high level query languages involves resolving issues such as:

- In what order should the database relations be scanned and by what access path?
- Which algorithms should be chosen for basic algebraic operations such as join and selection?
- When and where should temporary results be stored?

In centralized database systems, computation cost and secondary storage access cost dominate the cost of processing a query. The structure of query optimization is guided by the interactions between these cost components.

Many papers address the problem of optimization of individual queries for relational database systems [6,7,13]. A separate "access plan" is generated and executed for each query. The cost of processing a series of queries evaluated in this manner is equal to the sum of the processing costs for each query.

Recently, multiple-query optimization has been addressed in many papers [3,11,15]. A single access plan is generated for a series of queries to reduce the overall cost of processing the series of queries in comparison to the combined cost of processing each query from the series separately. The multiple-query optimization approach is attractive when:

- A series of queries is embedded in an application program such as EQUEL [16].
- A series of queries is submitted for batch processing.
- Queries are interactively submitted such that several may be pending at some point in time.
- Deductive query processing causes single deductive queries to generate a series of conventional queries.

The multiple-query evaluation process can be divided into the following steps:

[1] Decompose queries into high-level primitive operations (e.g., selection, join, projection).

[2] Identify common expressions and construct a global access plan.

[3] Choose the most efficient algorithm for each operation.

[4] Convert the series of queries into a lower-level program (perform query translation).

[5] Compile using whatever conventional optimization techniques are appropriate (e.g., loop jamming, register allocation, etc.).

Previous research on algorithm selection has centered around choosing the best algorithm for each basic algebraic operation (e.g., join, selection) considered independent of context [2]. In other words, the same algorithm is always used for the same type of operation. Some work has been done toward using information about the operands of each query (e.g., relation size) to choose among alternative algorithms for each operation type [4,13]. However, in this paper we propose that algorithms for each operation be selected by examining the entire series of queries and the interactions implied by particular algorithm choices.

Using the analysis of individual operations, it is relatively easy to determine for each operation a small series of feasible algorithms. Some of these algorithms are clearly inferior to others, regardless of context; these can be eliminated to create a series of possibly-optimal algorithms for each type of operation. The problem of optimal algorithm selection is then simply the choice of the proper algorithm from each series for each operation. We represent the differences between alternative possibly-optimal algorithms by the series of hashed temporary values (temporary relations) each algorithm uses and creates.

If a temporary relation is created in one operation and can be reused in a later operation, then the cost of recreating that temporary relation can be averted. This effect

can make the later operation be "cheaper" using the algorithm involving the temporary, whereas if the temporary actually had to be created rather than reused, an alternative algorithm would be cheaper. Some operations also have the effect of making temporaries become invalid because they modify the underlying relation, in such cases, one has the opportunity to update the temporary so that it may be reused (still possibly cheaper than recomputing the temporary relation) or the algorithm may simply invalidate the temporary relation (assuming that the temporary relation would not be used again). These interactions are at the core of the proposed algorithm selection procedure.

In section 2 a summary of relevant work is presented, discussing both previous work on multiple-query optimization and the hash-based algorithms typically used to implement individual operations. Section 3 presents the proposed algorithm choice optimization procedure. The interactions between previous multiple-query optimizations and the proposed algorithm choice procedure are outlined in section 4. Finally, section 5 summarizes the paper and indicates the direction of future research.

2. Related Work

In this section, previous multiple-query optimization techniques and hash based algorithms for individual operations are reviewed. Throughout the examples in this paper, queries are expressed in the tuple relational calculus language QUEL [16].

2.1. Multiple-Query Optimization

The objective in grouping a series of queries together is to reduce the total evaluation cost by identifying and eliminating common subexpressions. This goal is approached in two slightly different ways, one based simply on detecting common subexpressions given a particular query execution order, the other based on reordering the execution of queries so as to minimize the expected execution time (by inducing common subexpressions).

It is relatively straightforward to identify common subexpressions and implied relationships [11]. However, sharing of common expressions during execution is not always better than re-evaluation of the expressions (due to variations in the sizes of relations meeting particular constraints).

Example 1

Consider 3 relations:

```
EMP(name, salary, department)
SALES(department, item)
SUPPLY(item, supplier)
```

and two queries `Q1` and `Q2`:

```
Q1: retrieve EMP
    where SALES.item = 'radio'
        and EMP.salary > 20000
        and EMP.department = SALES.department
Q2: retrieve EMP
    where SALES.item = 'toy'
        and EMP.salary < 15000
        and EMP.department = SALES.department
```

In Q1 and Q2, sharing the common subexpression EMP.department = SALES.department might not be beneficial because early restrictions (e.g., selection operations) may substantially reduce the size of the relations which are involved in this join operation.

Also it should be noted that common subexpressions cannot be identified across update operations because the updated relation is treated as a completely new relation unrelated to the original relation. This problem is very similar to that which arises in disambiguating array references within conventional language compilers: storing a value into a[i] where the nothing is known at compile time about the value of i makes it necessary to assume that any element of a may have been changed [1].

The other approaches, which are based on reordering the execution of queries, are generally referred to as techniques for formulating a "global access plan" — an execution ordering of the subqueries within a series of queries. Typically, these systems do not reorder update operations, but only retrieve queries [15].

The approach consists of formulating several different access plans, computing expected evaluation costs for each, and then accepting the best of those tried [15]. This can also be treated as a dynamic programming problem [11].

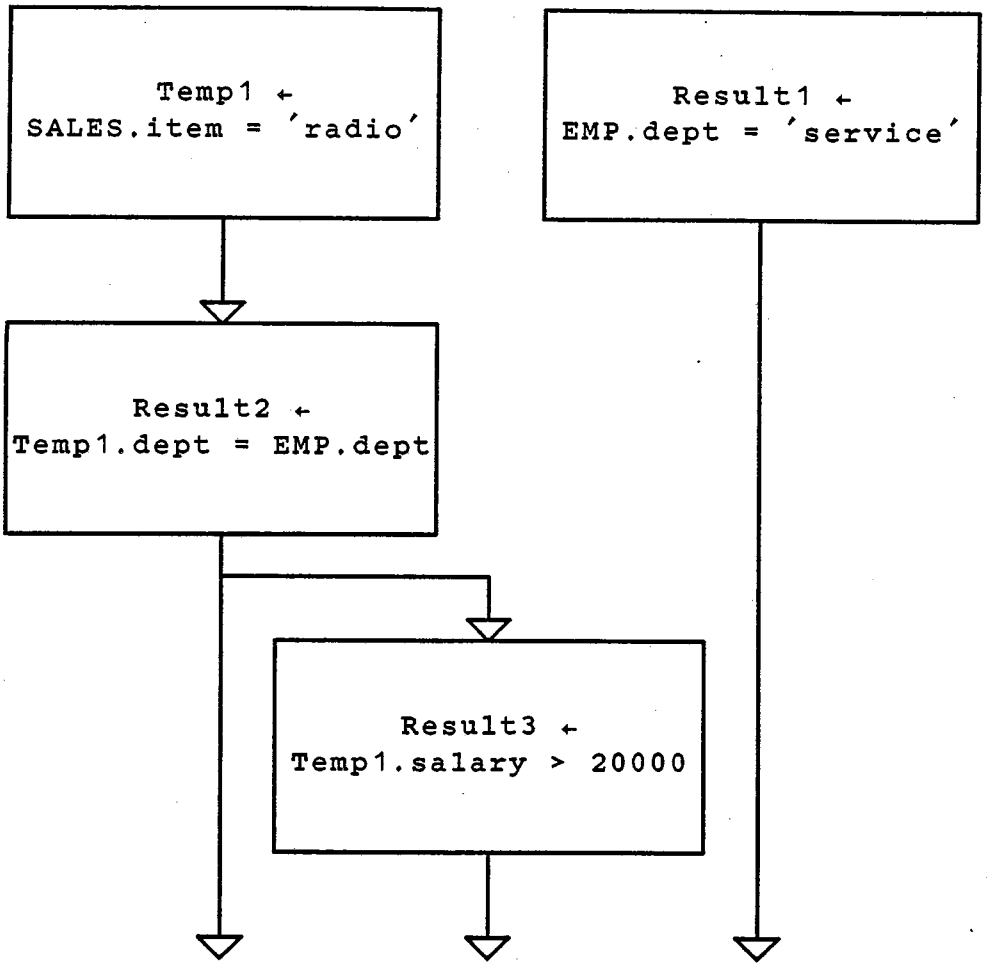
Example 2

Consider three queries using the same relations as in example 1:

```
Q1: retrieve EMP
    where EMP.dept = 'service'
Q2: retrieve EMP
    where SALES.item = 'radio'
        and SALES.dept = EMP.dept
Q3: retrieve EMP
    where SALES.item = 'radio'
        and EMP.salary > 20000
        and SALES.dept = EMP.dept
```

In this series of queries, the result of Q2 implies the result of Q3 and there are two common subexpressions which are SALES.item = 'radio' and SALES.dept = EMP.dept. But Q1 has neither common subexpressions nor implied relationships

with Q2 and Q3. Therefore the following access plan could be applied:



In order to distinguish between the multiple-query optimization described above and the multiple-query optimization we propose, we find it convenient to refer to the above as "operation-level optimization whereas the optimization presented in this paper is "algorithm-level optimization."

2.2. Algorithms for Relational Algebra Operations

It is generally accepted that there are three main types of algorithms for computing relational algebra operations. These differ primarily in the basic indexing structure used: linear scan using nested loop, a tree structured sort-merge, or a hash indexing scheme.

In general, as databases become large, the hash based algorithms tend to be most efficient; for example, the GAMMA [5] database machine uses only hash based algorithms. The remainder of this section outlines hash based algorithms implementing join, union, intersection, set difference, division, and selection[1].

[1] A hash based Cartesian product is not described because a linear scan using nested loops — the

For convenience, in the following examples which require relations, the three relations $R(a,b,c)$, $S(d,e,f)$, and $T(g,h,i)$ are used.

2.2.1. Algorithms Implementing Join

It is well known that one of the most costly operations in database processing is join. Several join algorithms (nested-loop, sort-merge [2], and hash based join [8]) have been studied and the performance of these alternatives has been compared [4,13]. There are two kinds of hash based join algorithms, split based and non-split based [10]. The split-based join algorithm divides relations into a number of disjoint subrelations before processing the join operation. The non-split join algorithm does not divides relations before processing. Which algorithm is best depends on the ratio between the size of relations which are involved in the join and the size of main memory [10].

If one assumes that relations are always very large, then the split-based join is always preferable. The split works because if a tuple x in R is in R_i and it joins with a tuple y in S, then the joining attributes of x and y must be equal. Therefore there is no joining of tuples across different hash buckets. This is why, to join R and S, it suffices to join the subsets R_i and S_i for each i.

The algorithm is roughly:

[1] Select a hash function h and partitions the values of h into, say, H_1, ..., H_n.

[2] Partition relation R into partition elements R_1, ..., R_n, where a tuple x in R is in R_i whenever $h(x.a)$ is in H_i where a is a hash attribute.

[3] Partition relation S into partition elements S_1, ..., S_n, where a tuple y in S is in S_i whenever $h(y.d)$ is in H_i where d is a hash attribute.

[4] Join R_i *and* S_i *for each* i in *1* to n.

It is convenient to represent these algorithms using a somewhat more compact abstract notation. In this notation, the above algorithm becomes:

[1] select hash-function h

[2] $[R_1, \ldots, R_n] \leftarrow h(R.a)$; a is the join attribute of R

[3] $[S_1, \ldots, S_n] \leftarrow h(S.d)$

[4] for each i in *1* to n do
    join(R_i, S_i)

As hinted above, the apparently recursive invocation of *join*(R_i, S_i) simply indicates that the split-based join decomposes the relations into smaller relations which must be joined. Since these relations are so much smaller, it is not critical for our purposes exactly which join algorithm implements them.

---

obvious implementation — is generally the most efficient for this task.

2.3. Algorithms Implementing Selection

Because selection is generally viewed as straightforward and relatively inexpensive, little attention has been given to the analysis of alternative selection algorithms. However, selections are common in database programs and even a small improvement can be noticeable. Two selection algorithms are considered in this paper:

Loop Selection Algorithm:

Tuples are selected from a relation by testing each tuple against the selection condition and extracting only those which satisfy the condition. This algorithm is generally represented in the low-level form as a loop over all tuples in the relation.

Hash Selection Algorithm:

Hash selection operates much as the loop selection algorithm, but attempts to minimize the number of full tests against the selection condition by using a hash function to partition the relation so that the test loop need only be performed on the tuples within one partition element. The algorithm is (assuming that the condition is an equality):

[1] select hash-function h

[2] $[R_1, \ldots, R_n] \leftarrow h(R.a)$; a is the hash attribute of R

[3] $i \leftarrow h(selection\text{-}value)$

[4] $select(R_i.a = selection\text{-}value)$

Clearly, loop selection performs better than hash selection unless the hash function is significantly cheaper than the full condition test and only a small fraction of the tuples will be in the partition element requiring the full test. It is unlikely that hashing would be more efficient, since the full test is most often a simple comparison for equality.

However, if, as a side effect of computing some other high-level operation, there exists an appropriate partition element for the same hash attribute, then the hash selection algorithm is more efficient because it tests fewer tuples.

2.4. Algorithms Implementing Other Operations

Hash based union, intersection, and set difference algorithms work same as hash based join algorithm if $join(R_i, S_i)$ in step 4 of join algorithm is replaced by $union(R_i, S_i)$, $intersect(R_i, S_i)$, or $difference(R_i, S_i)$. Hash based division algorithm which *divide R by S* can be represented as follow:

[1] select hash-function h

[2] $[R_1, \ldots, R_n] \leftarrow h(R.a)$; a is the hash attribute of R

[3] $[S_1, \ldots, S_n] \leftarrow h(S.d)$

[4] for each i in 1 to n do
$T_i \leftarrow divide(R_i, S_i)$

[5] $intersect(T_1, \ldots, intersect(T_{n-1}, T_n))$

Note that *divide(R_i, S_i)* is the same as *join(R_i, S_i)* if the hash function perfectly partitions the divisor relation *S*. Hash based algorithm can be used for projection at duplicate elimination stage. [2]

3. Algorithm-Level Optimization

After the operation-level optimization has been performed and algorithms for these operations have been selected, queries are converted into a lower level representation of the program. This lower level form is essentially a program or intermediate code structure with only operations close to the machine instruction level represented, i.e., database operations do not appear as single items, but as sequences of instructions which implement the database operations.

Since the database program in this form is indistinguishable from a conventional program, conventional compiler optimization techniques can be applied at this stage. For example, if two selections on a relation are converted into two low level loops with identical ranges, then the compiler can apply loop jamming [1] to remove the overhead of one of the loops. Note that loop jamming would not have been applicable to the original form of the query program. This is also true of common subexpression elimination, instruction scheduling, etc.

Conversely, current compiler transformation technology does not provide the ability to recognize an algorithm and to select a dramatically different alternative algorithm — yet this is easily done at the higher operation level. In addition, since the compiler operating on the low level form has no concept of a "relation," it is unable to figure out the effect of changes to relations. For example, let `Temp` be a temporary relation that is a subset of relation `R` such a particular property holds. The low level form cannot represent the fact that the tuples in `Temp` still maintain the desired property after a change has been made to `Temp`. However, high level optimization might use `Temp` instead of `R` in computing a further-constrained relation.

To span the gap between operation-level and traditional compiler optimization, we propose "algorithm-level optimization." The algorithm-level optimizer does not optimize high-level operations nor does it optimize low-level code in the traditional optimizing compiler sense, rather, it uses information from both levels to intelligently decide which algorithm should be used to represent each occurrence of a high-level operation in the low-level form.

A prime consideration in the choice of algorithm for each high-level operation is the amount of the computation which is not redundant with other computations. In other words, one algorithm may be cheapest when considered by itself, but if a partial result is available, a different algorithm may be cheapest. The availability of these partial results,

[2] For completeness, we have included these operations in our discussion although space did not permit including them in the algorithm choice procedure given in the appendix.

or "common subexpressions," is determined by predicting what temporary relations will exist in later operations. Unlike low-level optimizers, the algorithm-level optimizer knows what effect high-level operations may have on relations, hence it can be applied to queries that contain update operations (i.e., append, delete, and replace) as well as to those which simply retrieve values.

It should be noted that the algorithm-level optimizer tracks "temporary relations" which are created *within* individual high-level operations and that these relations are not directly represented either in the high-level or low-level forms. For example, each hash partition of a relation can be viewed as a temporary relation in this sense.

3.1. Update Operations

We consider three update operations: delete, append, and replace. As for the retrieve operation, each operation specifies a relation to be updated and a qualification clause specifying which tuple of the relation are to be affected. We term two kinds of qualification clauses simple and complex. A complex qualification requires examination of multiple relations in order to determine the tuples to be updated. A simple qualification is one which involves only one relation and, hence, which can always be performed in a single scan of the relation.

3.1.1. Delete

The delete operation simply removes all selected tuples, hence, a delete with a simple qualification requires no temporary relations and is not amenable to algorithm-level optimization. However, complex qualifiers used in delete operations effectively embed join operations and algorithm-level optimizations can be applied.

Although the fastest way to determine the set of affected tuples for a complex qualifier delete operation is using hash partitions, there are two techniques for this. In one case, the hash partitioning is used only for the delete operation — the partition is not updated to reflect the delete, hence it is generally not useful after the delete. In the other case, the partition is updated to reflect the relation with the tuples deleted, hence, the partition remains valid after the operation. The first method is better than the second if no operations following the delete require hash partitions with the same hash attribute. However, if further operations can use the same hash attribute, the second technique saves the expense of rehashing the entire relation. For example:

```
delete R
  where R.a = S.d
```

can be represented in abstract form as:

[R_1, ..., R_n] ← *h(R.a)* ; *a* is the hash attribute of *R*
[S_1, ..., S_n] ← *h(S.d)*
for each *i* in *1* to *n* do
  delete(R_i, join(R_i, S_i))
    ; delete tuples from *R* and R_i which satisfy
    ; the condition *join(R_i, S_i)*

Notice that all tuples which remain after the deletion are still validly partitioned according to the hash function. Hence, if a later operation can also make use of partitioning by this hash function, the re-partitioning of the relation is unnecessary. For example, if the next operation was:

```
retrieve R
  where R.a = 'abc'
```

then the complete computation would simply be:

[R_1, ..., R_n] ← *h(R.a)* ; *a* is the hash attribute of *R*
[S_1, ..., S_n] ← *h(S.d)*
for each *i* in *1* to *n* do
  delete(R_i, join(R_i, S_i))
    ; delete tuples from *R* and R_i which satisfy
    ; the condition *join(R_i, S_i)*
i ← *h('abc')*
retrieve(R_i, select(R_i.a = 'abc'))

which avoids the recomputation of the hash partition for the relation *R*.

3.1.2. Append

Append is used to add new tuples to a relation. The new tuples can either be explicitly given in the append command or they can be derived by performing a specified operation to a set of tuples selected from an existing relation. We call these two kinds of append "explicit" and "derived." Explicit appends are far more common than derived appends.

An explicit append is specified by giving the name of the relation and a list of tuple values to be appended to that relation. In general, it is preferable that if a tuple to be appended exactly matches a tuple which exists in the target relation then the tuple is not appended — duplicate tuples are not created. In order to avoid making duplicate tuples, it is necessary to scan the target relation for copies of the tuples to be appended. This can be accomplished either using a hash partition of the target relation or by linearly scanning the target relation itself.

A derived append specifies the target relation, a clause which selects a set of tuples from which the new tuple set will be derived, and the operations to be performed to derive the new tuple set. Just as for the delete operation, the clause which selects a set of

tuples can be simple or complex (using the same definitions as for delete).

A derived append with a simple selection clause can be implemented very much as an explicit append, except in that the tuples being selected may be taken from a relation other than the target relation.

However, a derived append with a complex selection clause is somewhat more profound than a complex delete. Suppose that a new relation *R(a,b,c)* is to be created using a derived append with a complex clause involving relations *S(d,e,f)* and *T(g,h,i)*. For example:

```
append to R ( a=S.d, b=T.h, c=S.f+T.h )
  where S.d = T.g
```

In this example, only the *S.d*, *T.h*, and *S.f* fields are used in deriving the tuples to be appended. Hence, the implicit join can actually be filtered by a projection of the fields used in the operations creating the new tuples, in this case, a projection of *d*, *f*, and *h* over the temporary relation. Further, although the operations creating the new tuples are specified in terms of the original relation field names, these references would have have to be internally translated to reference the fields of the temporary relation resulting from the join.

If there are further clauses using the same condition, then the projection used should be over the union of the fields needed by the entire sequence of such operations. This is actually a multiple-query optimization, noted here simply because it is not generally discussed in multiple-query optimization.

3.1.3. Replace

Although the replace operation is often treated as a delete followed by append, however, the append operation is potentially far more complex than replace. This is because the replace qualification clause is constrained to refer to the target relation: only existing tuples can be replaced.

For this reason, replace is most similar to a delete operation, differing only in that the selected tuples are modified rather than deleted.

3.2. Algorithm Choice

As discussed above, many of the basic database operations (not just retrieve operations) are implementable by any of several algorithms and proper handling of interactions between algorithms for a series of operations can yield significant performance improvements. In this section, we attempt to formalize the way in which optimal algorithm choices can be made. There are several steps before the optimization algorithm can be applied.

3.2.1. Preparatory Steps

First, each query is separated into two steps: qualification and effect. The qualification step selects tuples which satisfy the qualification clause of the query. The effect step performs the operation specified on the tuples which were selected in the qualification step. For our purpose, retrieve operations are considered to have no effect step since they do not change any relation; further, simple delete, simple replace, and explicit append operations are considered to have no qualification step.

Second, either a local or global access plan is established. This was described in section 2.1.

Finally, in the third preparatory step, queries are transformed into an intermediate form which will be the input to the algorithm-choice analysis procedure. This intermediate form includes predictions for the costs incurred if hash partitions need to be updated.

For the computation of costs, the following table defines the relevant variables.

comp	time for comparing keys in main memory
hash	time to hash an attribute which is in main memory
move	time to move a tuple in main memory
IO	time to read or write a block between disk and main memory
n	number of hash partitions for a given relation
|R|	number of pages in R (similar for S and T)
{R}	number of tuples in R (similar for S and T)

Suppose temporary relation *Temp* is appended to relation *R*, then the approximate cost will be:

```
{Temp} * (hash + move)          ; hash tuple and move to
                                ; output buffer
+ |Temp| * 2 * IO               ; read and write partitioned
                                ; relation from and to disk
```

and *cost_of_rehashing_relation* (the relation *R*) at that time will be:

```
{New_R} * (hash + move)         ; hash tuple and move to
                                ; output buffer
+ |New_R| * 2 * IO              ; read and write partitioned
                                ; relation from and to disk
```

where *New_R* is $R \cup Temp$.

The cost of delete depends on the type of deletion. In the case of a complex delete, if the hash partitions can be updated while the qualification clause is being evaluated (e.g., change hash partitions for *R.a* with *delete R where R.a = S.d*) then the cost is simply

|*New_R*| * *IO* where *New_R* is *R* - *Temp*. But if temporary relation *Temp* is deleted from relation *R* **which already has hash partitions**, then the approximate cost will be:

```
{Temp} * (hash + move)                  ; hash tuple and move to
                                        ; output buffer
+ ({Temp}/n * {R}/n) * n * comp         ; probe for a match
+ (|R| + |Temp|) * IO                   ; read relations from disk
+ |New_R| * IO                          ; write relation to disk
```

This exceeds *cost_of_rehashing_relation* (i.e., {*New_R*} * *(hash + move)* + |*New_R*| * *2* * *IO*), hence it is not necessary to consider the possibility of using a hashing scheme.

In the case of simple delete, the cost of delete ranges from one partition element being changed (e.g., change hash partitions for *R.a* with *delete R where R.a = 'abc'*) to all partition elements being changed (e.g., change hash partitions for *R.a* with *delete R where R.b = 'cde'*). If all partition elements must be changed, the cost of hash delete exceeds *cost_of_rehashing_relation.*

The cost of replace can be thought of as the sum of the cost of delete plus the cost of append operation.

3.2.2. The Algorithm Selection Procedure

In the algorithm there are three types of operations:

[1] those which do not require hash partitions, but which can be efficiently implemented using hash partitions if they are available (e.g., selection)

[2] those which require hash partitions which are then modified, with the algorithm choice being whether to update the hash partition or simply to update the base relation (e.g., delete, append, replace)

[3] and those for which there is only one good algorithm, which always produces hash partitions (e.g., join)

We call each potentially-generated hash partition an "available node" (*anode*). When it cannot be determined in one pass (without lookahead) whether the optimum algorithm is the choice which generates a particular hash partition, the algorithm selection procedure produces a "conditional" available node. Later, when sufficient lookahead has been obtained to decide that these conditional anodes should have been generated, the anodes are made unconditional. After the entire input has been processed, any remaining conditional anodes indicate that the corresponding algorithms should not be applied (i.e., the alternative algorithms should be used). Hence, a second pass can directly encode the selected algorithms for the entire input.

A detailed description of algorithm selection procedure, PreAvail, is presented in appendix.

3.2.3. Incremental Application

It is significant that this procedure also can be incrementally applied to queries in an interactive query environment.

Suppose that the system is given, in real time, a series of queries Q, which at time T_0 is $\{q_1, q_2, q_3, \ldots, q_n\}$. The above algorithm selection procedure is applied, hence generating a table of anodes representing the optimal algorithm choices given the queries available at time T_0. Hence, the system can now initiate the computation of q_1; at this time, any remaining conditional anodes for, or linked to, q_1 are deleted.

While q_1 is executing, time passes and new queries are added to the system. Suppose that at time $T_{0+\delta}$ one additional query is made, giving the sequence $\{q_2, q_3, q_4, \ldots, q_n, q_{n+1}\}$. All the existing anodes are still valid; only the anodes for query q_{n+1} must be evaluated. Notice, however, that the evaluation of the anodes for q_{n+1} may cause earlier conditional anodes become unconditional.

Hence, using simple incremental application of the algorithm selection procedure, the algorithm choice for earlier, but not yet executed, operations can be optimized on the basis of all enqueued queries.

3.2.4. Other Considerations

As stated above, the principle upon which algorithm selection is based is the availability and reuse costs associated with hashed temporary relations.

In computing these costs, we have not considered the use of various physical data organizations (data layouts) which may differ from system to system. We further assumed that there is no limit on the number of simultaneously live temporary relations. However, these restrictions could be lifted with only minor adjustment to the algorithm selection procedure.

Although we presented the computation of simple, fixed, costs, the procedure does not change if the costs are variables; such cost computations are simply beyond the scope of this paper. If only a limited number of simultaneously live temporary relations can be accommodated, then the problem of selecting which should be created is equivalent to that of register allocation and assignment within conventional compilers. Hence, any of the usual register allocation techniques can be used as a second pass to correct algorithm choices which created too many temporaries.

4. Interaction Between Algorithm-Level And Operation-Level Optimization

So far, the algorithm-level optimization has been applied to fixed access plan. But better benefit can be obtained if global access plan considers the effect of the algorithm-level optimization. Let's consider the following example:

```
Q1: retrieve R
    where R.a = 'abc' and R.b = 'def'
```

can have two access plans P1 or P2:

```
P1: Temp1 ← R.a = 'abc'
    result ← Temp1.b = 'def'
```

or

```
P2: Temp1 ← R.b = 'def'
    result ← Temp1.a = 'abc'.
```

If one of the hash partitions of `R.a` or `R.b` already exists then one plan is better than the other. If the global access plan analysis generates two apparently equivalent access plans `P1` and `P2` then algorithm-level optimizer will select one of them based on performance at that level. For example:

```
Q1: retrieve R
    where R.a = 'abc'
Q2: retrieve R
    where R.a = S.d and R.b = S.e
Q3: retrieve T
    where R.b = T.g
```

will produce either access plan `P1`:

```
P1: result1 ← R.a = 'abc'
    Temp1 ← R where R.a = S.d
    Result2 ← Temp1 where Temp1.b = S.e
    Result3 ← T where R.b = T.g
```

or `P2`:

```
P2: result1 ← R.a = 'abc'
    Temp1 ← R where R.b = S.e
    Result2 ← Temp1 where Temp1.a = S.d
    Result3 ← T where R.b = T.g
```

If plan `P1` is chosen, then hash partitions of *R.a* are shared between selection of *Q1* and join of *Q2*; if plan `P2` is chosen then joins of *Q2* and *Q3* share hash partitions of *R.b*. This leads to the following evaluation:

```
benefit of one extra selection operation (P1):
((n-1)/n) * ({R} * comp + |R| * IO)

benefit of one extra join operation (P2):
{R} * (hash + move) + 2 * |R| * IO
```

Hence, since plan P2 is expected to be faster, it will be chosen.

5. Conclusions And Future Work

This paper shows that significant improvements can be gained by careful choice of the algorithm to be used for each operation. This choice is not merely based on efficiency of algorithms for individual operations, but rather on the efficiency of the algorithm choices for the entire multiple-query evaluation. An efficient procedure for automatically optimizing these algorithm choices is given.

The interaction between the algorithm-level optimizer and operation-level optimizer further demonstrates the potential improvements to be gained by the integration of both optimization levels.

Although the algorithm selection procedure is given using hash based algorithms, we believe that the same general approach can be applied to a far wider range of algorithms. There is also very little linking the procedure to a particular query language, and the technique should easily apply to other very high level languages such as SETL [14] and PROQUEL [9].

Ongoing research centers on the parallelization and optimization of multiple queries for multiprocessor systems.

References

[1] Aho, A. V., and Ullman, J. D. Principles of compiler design. Addison-Wesley (1977)

[2] Blasgen, M. W., and Eswaran, K. P. Storage and access in relational databases. IBM Syst. J. 16, 4 (1977)

[3] Chakravarthy, U. S., and Minker, J. Multiple query processing in deductive database using query graphs. Proceedings of the Conference on Very Large Data Bases (1986)

[4] Dewitt, D., and Gerber, R. Multiprocessor hash based join algorithms. Proceedings of the Conference on Very Large Data Bases (1985)

[5] Dewitt, D., et al. GAMMA - A high performance dataflow database machine. Proceedings of the Conference on Very Large Data Bases (1986)

[6] Finkelstein, S. Common expression analysis in database applications. Proceedings of the ACM-SIGMOD International Conference on Management of Data, (1982)

[7] Jarke, M., and Koch, J. Query optimization in database systems. ACM Computing Surveys, 16, 2 (June 1984)

[8] Kitsuregawa, M., et al. Application of hash to database machine and its architecture. New Generation Computing. 1 (1983)

[9] Lingat, J., et al. Rapid application prototyping the PROQUEL language. Proceedings of the Conference on Very Large Data Bases (1988)

[10] Nakayama, M., et al. Hash-partitioned join method using dynamic destaging strategy. Proceedings of the Conference on Very Large Data Bases (1988)

[11] Park, J., and Segev, A. Using common subexpressions to optimize multiple queries. Proceedings of Conference on Data Engineering (1988)

[12] Rosenkrantz, D. J., and Hunt, H. B. Processing conjunctive predicates and queries. Proceedings of the Conference on Very Large Data Bases (1980)

[13] Shapiro, L. D. Join processing in database systems with large main memories. ACM Transactions on Database Systems, 11, 3 (1986)

[14] Schwartz, J., et al. Programming with sets: An introduction to SETL. Springer-Verlag (1986)

[15] Sellis, T. K. Multiple-query optimization. ACM Transaction on Database Systems, 13, 1 (1988)

[16] Stonebraker, M., et al. The design and implementation of INGRES. ACM Transactions on Database Systems, 1, 3 (1976)

Appendix

```
/*
  Algorithm: PreAvail

    Precompute availability of hashed temporaries so that
    hash based algorithms will be applied only when either
    the hash algorithm is always superior or the hash
    algorithm generates as temporary which can be profitably
    reused later in the multiple-query sequence.

  Given input of:
      join(R, a, S, d, 0)
      selection(R, a, null, null, 0)
      delete(R, null, R, a, cost_of_delete)
      replace(R, null, R, a, cost_of_replace)
      append(R, null,  null, null, cost_of_append)

  with the generalized form:
      op(r1, a1, r2, a2, opcost)

  Constructs output consisting of a list of:
      anode(rel, att, op, cond, cost, avail, pointer)

  where:
    anode:        available node structure
    rel:          relation
    att:          attribute
    op:           database operations
    condition:    either cond for conditional or
                  uncond for unconditional
    cost:         the cost of modifying hash partitions
    availability: either true or false
    pointer:      pointer to previous conditional anode

  July 1989 by Myong Kang & Hank Dietz
*/

PreAvail()
{
  for (i is each high-level operation) {
    Process(i);
  }
}
```

```
Process(i)
{
  switch (i.op) {
  case selection:
    t = anode(i.r1, i.a1, _, _, _, true, _);
    if (!t) {
      create(anode(i.r1, i.a1, selection, cond, 0, true, null));
    }
    break;
  case join:
    hash(i.r1, i.a1);
    hash(i.r2, i.a2);
    break;
  case append:
    append(i.r1, i.opcost);
    break;
  case delete:
  case replace:
    delete_replace(i.r1, i.r2, i.a2, i.op, i.opcost);
  }
}

hash(rel,att)
{
  a = anode(rel, att, _, _, _, true, _);
  if (a) {
    if (a.condition == cond) {
      do {
        a.condition = uncond;
        a = a.pointer;
      } while (a != null);
    }
  } else {
    create(anode(rel, att, join, uncond, 0, true, null));
  }
}
```

```
append(relation, mod_cost)
{
  for (a is each anode(relation, _, _, _, _, true, _)) {
    a.availability = false;
    if (a.condition == uncond) {
      create(anode(a.rel, a.att, append, cond,
                                    mod_cost, true, null));
    } else {
      new_cost = a.cost + mod_cost;
      if (new_cost < cost_of_rehashing_relation) {
        create(anode(a.rel, a.att, append, cond,
                                      new_cost, true, a));
      }
    }
  }
}

delete_replace(rel1,rel2,att2,op,mod_cost)
{
  for(t is each anode(_, _, selection, _, _, true, _)) {
    t.availability = false;
  }

  if (rel1 == rel2)  {
    a = anode(rel2, att2, _, _, _, true, _);
    if (a) {
      a.availability = false;
      if (a.condition == uncond) {
        t = create(anode(rel2, att2, op, cond,
                                  mod_cost, true, null));
      }
      else {
        new_cost = a.cost + mod_cost;
        if (new_cost < cost_of_rehashing_relation) {
          t = create(anode(rel2, att2, op, cond,
                                    new_cost, true, a));
        }
      }
    }
  }

  for (a is each anode(rel1, _, _, _, _, true, _) && a != t) {
    a.availability = false;
  }
}
```