OPTIMIZATION OF EXTENDED DATABASE QUERY LANGUAGES

by

T. K. Sellis and L. Shapiro

Memorandum No. UCB/ERL M85/1

18 January 1985

OPTIMIZATION OF EXTENDED DATABASE QUERY LANGUAGES

by

T. K. Sellis and L. Shapiro

Memorandum No. UCB/ERL M85/1

18 January 1985

title page

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

OPTIMIZATION OF EXTENDED DATABASE QUERY LANGUAGES

Timos K. Sellis
Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

Leonard Shapiro
Department of Computer Science
North Dakota State University
Fargo, ND 58105

Abstract

In this paper we examine the problem of query optimization for extended data manipulation languages. We propose a set of tactics that can be used in optimizing sequences of data base operations and describe the corresponding transformation procedures. These transformations result in new equivalent sequences with better space and time performance. The proposed techniques are especially useful in artificial intelligence and engineering applications where sequences of commands are executed over high volume databases.

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 83-0021 and by the Naval Electronics Systems Command Contract N00039-78-G-0013.

1. Introduction

Existing database languages, including QUEL ([STON76]) and SQL ([CHAM76]), were designed to meet the needs of business data processing applications. That generation of languages, and their associated database management systems, are now reaching maturity.

Benefits of DBMSs and languages include efficient access to large disk-based data sets and a nonprocedural interface to data. Disk-based data, and complex programs requiring a higher level programming interface to data, are now beginning to be common in other applications areas, such as artificial intelligence and engineering design automation.

To extend the benefits of the data base approach to other areas, many researchers have defined various extensions to existing database languages. Examples of these extended languages include the language QUEL* ([KUNG84]), designed to support artificial intelligence applications, GEM ([ZANI83]), to support a semantic data model, and the proposal of ([GUTT84]), for support of VLSI design.

As with the introduction of relational DBMSs, the first attempt at implementing these extended query languages has resulted in terrible inefficiencies, e.g. several minutes to search a path through a 30 by 30 node map ([KUNG84]). Relational DBMSs were made efficient largely through the use of sophisticated optimization algorithms ([WONG76,SELI79]). In this paper we propose an extension of those optimization algorithms to the new extended query languages. Although we use QUEL* as an example, the principles we propose should be applicable to a wide variety of extended languages.

In this paper we will consider only the case of programs consisting of database commands embedded in a host language, although our results also apply to interactive commands. It is common to optimize each query in a program separately. To "optimize" a query means to choose among the various ways of executing the query. For example, there may be a choice of indexes to use, or a choice of strategies for executing a relational operator such as the join. In ([FINK82]) it is proposed that some interquery optimization be done. We extend those ideas here. We know of no actual implementation which does interquery optimization.

The paper is organized as follows: In the next section we present the extended language QUEL* and define the optimization unit. In section 3 we describe various optimization tactics for use by a QUEL* optimizer. Each of these tactics is related to a tactic from some other area, namely compiler construction, query optimization, or physical database design. In Section 4 we present two new optimization tactics, each of which transforms a sequence of QUEL* commands into a single REPLACE command. Finally, section 5 contains our conclusions and ideas for future research in the area.

2. What is Optimization ?

Optimization in database languages means to choose among the various ways of executing a command or set of commands in the language. For example, there may be a choice of indexes to use, or a choice of strategies for executing a relational operator such as the join. In this section we will examine what optimization will mean for extended languages. We begin the motivation of our definition of optimization by reviewing the structure of QUEL*.

2.1. QUEL*

QUEL* is an extension of QUEL, designed so that it is possible to code relatively sophisticated algorithms in QUEL* with little or no need for statements in a host programming language such as C. QUEL* adds to QUEL two new constructs, namely transitive closures of some QUEL commands and an EXEC command.

In QUEL*, by a transitive closure of a command we mean repeated application of that command until the database does not change. For example, the following command makes Joe the manager of every employee who eventually reports to Joe via the hierarchy of managers (the transitive closure of a command is denoted by the command followed by a "*"):

```
REPLACE* emp (mgr = "Joe")
         where emp.mgr = emp1.name
         and   emp1.mgr = "Joe"
```

The addition of transitive closure operators to QUEL was first proposed in [GUTT84], where it was identified as crucial to VLSI design applications. The APPEND* command especially behaves in the same way the Least Fixed Point operator does in [AHO79b]. For example, the APPEND* command makes QUEL* capable of computing the transitive closure of a relation.

The second new construct in QUEL* is the EXEC command, which executes a sequence of QUEL* commands. This EXEC command is similar to a suggestion in [STON84] which gives recursive power to QUEL by allowing the system to execute relation fields. It is most useful in its EXEC* form, when the given sequence of QUEL* commands is executed repeatedly, until the database does not change. For example, given a map with costs between neighbouring points described by a relation FEASIBLE (source,dest,cost) and a relation STATES (dest,cost) which will give for every point in the map its current cost from some specific START point, the following code represents an algorithm that finds the shortest-path from START to FINISH :

```
RETRIEVE into STATES (dest = START, cost = 0)

range of s,t is STATES
range of f is FEASIBLE

EXEC*
{
  APPEND to STATES (dest = f.dest, cost = f.cost+s.cost)
           where s.dest = f.source

  DELETE s
          where s.dest = t.dest and s.cost > t.cost
          or     s.cost > t.cost and t.dest = FINISH
}
```

2.2. What is an optimization unit?

Each new command in QUEL* represents a sequence of one or more QUEL commands. This is also true of the other extended database languages we have mentioned. In Guttman's thesis ([GUTT84]) the only new constructs are the * operations of QUEL*, and GEM has been implemented on top of INGRES ([TSUR84]). A primary difference between business data processing and other applications areas is that in the business data processing environment data base transactions are short, or involve one complex command, whereas in the domains for which extended database languages have been designed, transactions are lengthy and complex. Since a command in an extended language typically represents several commands in a classical database language, in this section we will propose that our query optimizer operate on a sequence of commands rather than the traditional approach of optimizing a single command at a time.

As a first attempt at designing a QUEL* optimizer, we could merely optimize each corresponding QUEL command separately, using an existing QUEL optimizer. For example, a REPLACE* command would be processed by generating one REPLACE command, optimizing and executing it, and continuing until the execution of the REPLACE command does not change the database.

We use the term "optimization unit" to refer to the unit acted on by the optimizer. Thus in QUEL the optimization unit is a single QUEL command ([STON76]).

We propose that for QUEL* the optimization unit will be a single QUEL* command, including even an EXEC or EXEC* command. With our proposal we have effectively made the optimization unit equal to any sequence of QUEL* commands, for any such sequence can be the argument of an EXEC command. In fact, if the programmer wishes, he can code an entire QUEL* program (containing no programming language commands) inside a single EXEC statment and the optimization unit will then be that entire program. There are at least two advantages to enlarging the optimization unit:

(1) The optimizer has more information on which to base its decision. For example, knowing that there will be several consecutive REPLACE commands executed, the optimizer may elect to build an index which is not worthwhile for only one REPLACE.

(2) The optimizer has more flexibility to rearrange the order and implementation of operations. For example, in an EXEC* which includes a DELETE command, it will be useful to do the DELETE operations as early as possible, in order to reduce the size of the relation to be processed.

There are also at least two possible disadvantages to this approach:

(1) Concerning the notion of transaction, we note that an optimization unit must be smaller than or equal to a transaction unit. This is because the optimizer may completely rearrange the order of execution of commands in an optimization unit. If there were an *end-transaction* statement inside the optimization unit, it would have a completely different meaning after a rearrangement. We know that as the size of transactions grows, the degree of concurrency decreases. In our context this means that as the optimization unit grows the concurrency will decrease. On the other hand, concurrency control is not a significant issue in many applications (such as artificial intelligence) for which these extended query languages are defined. In engineering design applications the most significant issue is how to handle very long transactions which occurs as a result of the nature of design (e.g. checking out a chip design for several days). Therefore increasing the size of the transaction unit may not be a significant disadvantage for extended database languages.

(2) As the size of the optimization unit grows, so does the complexity of the optimization task. The first comprehensive approach to query optimization ([WONG76]) proposed query decomposition as a method to avoid searching the exponentially growing space of query processing strategies. However, the most successful query optimization method has been that of System R ([SELI79]), which does perform essentially an exhaustive search of the strategy space. Even System R's strategy avoids searching the full strategy space by, for example, considering joins of at most four relations. Therefore if we allow the optimization unit to grow arbitrarily, the cost of searching the strategy space may exceed the savings in efficiency.

The benefit of these advantages, and the cost of the disadvantages, grows with the size of the optimization unit. The size of the optimization unit is to a significant extent under the control of the programmer, who can enlarge it by placing several QUEL* commands inside an EXEC command.

3. Optimization Techniques

In the previous section we proposed that an optimization unit be any QUEL* command, including an EXEC (which can include an arbitrary string of QUEL* commands). We claimed two advantages for this enlarged optimization unit, namely that the optimizer has more information and more flexibility. In this section we list specific techniques to make use of this added information and flexibility.

The three classes of optimization tactics we present in this section are each closely related to techniques used in other contexts, namely compiler design, query optimization and physical data base design.

In many of these tactics the size of relations is an important factor. Most of the tactics can be implemented as part of either a compiler or interpreter. In the compiler case, statistics must be estimated using methods analogous to those used in System R ([SELI79]).

3.1. Compiler Design Techniques

Optimization techniques in compiler design focus especially on two areas ([AHO79])

- loop optimization (time), and

- temporary storage management (space)

3.1.1. Loop Optimization

In data base commands loops are found in two levels, single queries and transitive closure (*) operations. The loops of the first kind are inherent in the commands. For example, a query involving a join between two relations can be implemented with a nested loop. On the other hand * operations are explicitly user defined loops.

The case of implicit loops has been studied in the past as the problem of finding execution plans that minimize execution time and avoid calculating the same expressions many times ([BLAS76,EPST79]). This corresponds to identifying loop invariants in compiler design.

In the context of * operations a new problem arises, namely the problem of identifying loop invariants within a set of commands. For example, aggregate computations that involve relations not updated during the execution of one iteration can be evaluated outside the loop and be replaced with a constant in the body of the loop. This resembles the previous case of intraquery optimization and the gain in execution time is substantial especially in cases where the result of the aggregate is involved in join clauses. An algorithm that does this transformation can be very easily derived for each aggregate in the loop, it will check if it involves relations that are not updated before this aggregate is encountered in the loop. If this is the case then the aggregate can be computed outside the loop and stored in a variable which then replaces every occurrence of the aggregate in the loop.

A more careful treatment of aggregates in loops is also possible if after doing the modifications suggested above there are still aggregates to be calculated in every iteration. It may be worth incrementally computing those aggregates, i.e. computing them once in advance and then every time the data involved changes after an update operation. For example, consider a query that needs to compute the average salary of all employees. We can define a "variable" AVG that will hold the result of this aggregate. Then, given an operation that inserts k new tuples with salaries $sal_1, sal_2, \ldots, sal_k$, the new value for AVG is computed using the formula

$$AVG = \frac{AVG*NUM + \sum_{i=1}^{k} sal_i}{NUM + k}$$

where NUM is the number of tuples in the relation. Similar formulas can be derived for all common aggregates, like MIN, MAX, SUM and COUNT. This technique will usually result in a more efficient implementation, if the number of inserted or deleted tuples is small. Although it will not work in all cases, it is not difficult to identify the aggregates for which it will work. Aggregates with no qualification part or with qualifications that are time invariant (do not include relations that are modified through out the execution of the loop) are clearly good candidates for this optimization.

Another common example of loop invariants is common subexpressions. For example, suppose we are given a sequence of operations on the EMPLOYEE relation where all qualifications restrict the initial relation to the set of tuples of employees working for Joe. Then it might be more efficient to create a temporary relation in advance that will contain only the tuples of those employees. This problem is examined in more detail in the following section because it is not found only in loops but generally in any given sequence of commands and is viewed as the problem of temporary storage management.

3.1.2. Temporary Storage Management

The problem of temporary storage management in the context of data base operations is how to optimize commands by reusing results (e.g. temporary relations) from the execution of preceding commands. In compiler design the same problem is found as the common subexpression problem ([AHO79]).

The problem can be attacked from two different directions. First, one can build a sophisticated caching scheme where temporary results are saved and can be used later in the execution of other commands (see [FINK82] for a detailed discussion of this approach). This scheme must be sophisticated since indexing of queries and clever validity and utility factors for the implementation of the cache must be invented. In [FINK82] the problem of identifying useful temporary relations that are saved in the cache is examined using the query graph representation for queries.

In this paper we describe a second approach, which is performed at *compile time*. This is contrasted to the *run time* approach that caching corresponds to. The problem that we examine can be stated as follows :

Given a sequence of QUEL commands is there a set of temporary relations that can be constructed and when should each of them be created so that the execution is more efficient in terms of temporary space usage and execution time.

For retrievals the above problem can be restated as rearranging the sequence of queries and arranging the construction and saving of the temporaries so that the new sequence of queries is equivalent (produces the same set of answers) but more efficient then the inital one. We will briefly refer now to this second problem.

As it was stated in the previous section, there is much analysis that can be done if a sequence of data base commands is given in advance. In conventional query processing, temporary relations are created depending on the query optimization methods used. It is the case that some of these temporaries are known that will be created well in advance. For example, temporary relations in INGRES are known except in the case of tuple substitution where the size of the relations is crucial and determines the outcome. We can observe that there are various ways to solve the temporary storage problem. First, one can use some normalized representation of queries (e.g. query graphs or tableaux) to examine how the execution of a query affects the execution of another query in terms of common subexpressions. For example, a simple minded approach would be to examine all possible permutations of the sequence in which the queries are executed and use the procedures from [FINK82] to determine what the total cost of execution would be. To give an example, the cost of executing

```
RETRIEVE (emp.all) where emp.age < 30
RETRIEVE (emp.all) where emp.age < 40
```

is much higher than the cost of executing

```
RETRIEVE (emp.all) where emp.age < 40
RETRIEVE (emp.all) where emp.age < 30
```

since in the latter case the second command can use the result of the first query.

The next step would be analogous to the System-R's query execution planner ([SELI79]), that is to build a decision tree and use some heuristic rules to prune down the size of the tree that is searched. The difference is that here we have a decision forest instead of a tree where each query is represented by a tree. Transforming this forest to another forest that guarantees more efficient execution, is a process that must preserve some chronological order on the sequence of the commands. Actually it is a problem similar to the serializability problem in concurrency control ([PAPA79]) where given the sequence of low level operations (joins, restrictions and projections) we would like to find a serial execution of the queries that corresponds to a nearly optimal sequence of the operations that is equivalent to the initial one. The major factors considered in this decision process are

- joins among relations are expensive operations
- restrictions are also expensive in the absence of indices or other fast access structures.

Therefore one would like to minimize first the number of times the same join operation is executed and then, using the restriction clauses, the number of times the same tuples are touched. Clearly there is a tradeoff between executing all joins first and then all the restrictions or interleaving join and restriction operations.

To extend further the previous idea, one might come up with a notion of a *distance* between queries. This distance will, somehow, measure the cost of executing one query given the execution pattern (intermediate temporary relations and results) of the other query. Having numerical values to express the complexity of executing a query allows us to use techniques from other areas in order to design efficient execution plans. For example, in [SELL82] the distances between characteristics of speech phonemes are used to define a potential function in the phonemes space. Then some notion of force is used to group characteristics of similar phonemes into clusters ([KACH82]) and a representative is selected for each cluster. In analogy, using similar algorithms we can identify clusters of queries which will be covered by the representative query. By "covered" we do not necessarily mean that the result of every member of the cluster is directly available from the result of the representative, but that the cost of getting this result from the result of the representative is much less than the cost of executing the whole query from the beginning. Finally, this technique can also be thought similar to reduction of single queries ([WONG76]) in conventional query processing where "loosely connected" subqueries within a single query are identified.

When update operations are involved in the sequence the above ideas are not directly applicable. In the next section we examine optimization techniques from query optimization where update commands are also considered.

3.2. Query Optimization Techniques

In 3.1.2 we examined a problem that is closely related to query optimization, namely the problem of finding a sequence of join, select and project operations that produces the most efficient execution pattern for a given query. In this section we examine some more ideas from query optimization that are useful in optimizing extended query language constructs. These are

- early restrictions

- combining operations

3.2.1. Early Restrictions

It is usually advantageous to restrict the size of the relations involved in a query as early as possible in the execution plan. For example, INGRES selects to execute all one variable selection clauses in the first step of processing. In QUEL*, DELETE commands can be thought as restrictions since they restrict the size of the relations involved in the subsequent commands. Therefore, one might want to incorporate the effects of DELETE commands as early as possible. Clearly this cannot happen always without changing the semantics of the program, but there are many cases where this transformation will be very useful. The effects of a DELETE command can be introduced by enchancing the qualifications of preceding commands. For example, the sequence

```
/* make Joe the manager of all employees */
APPEND to EMP (name=emp.name, salary=emp.salary, mgr="Joe")

/* but ... nobody can make more than his(her) manager */
DELETE emp
          where emp.salary  > emp1.salary
          and     emp1.name = emp.mgr
```

can be changed to

```
/* append only tuples of employees that make less than Joe */
APPEND to EMP (name=emp.name, salary=emp.salary, mgr="Joe")
          where emp.salary  <= emp1.salary
          and     emp1.name = "Joe"
```

```
/* and make sure nobody of the old employees make more than his(her)
   manager */
DELETE emp
        where emp.salary > emp1.salary
        and    emp1.name = emp.mgr
```

Notice how the above transformation resembles the implementation of integrity constraints using query modification ([STON75]). Doing this transformation, which is syntactic rather than semantic, does not require a lot of knowledge about the query and most of the time proves to be very useful, especially in cases where the number of tuples appended and immediately deleted by the next command, is large.

3.2.2. Combining Operations

In single query optimization one might prefer to execute both a selection and a join in a relation at the same time, to avoid scanning the same tuples twice. In our extended environment one might like, analogously, to combine the execution of multiple commands. In the case of RETRIEVE only commands, merging is possible and practical in many cases (the previous section examined this problem).

Consider now a sequence of two REPLACE commands. We show in the Appendix that there is a single REPLACE that produces the same result. This new command is the composition of the two previous commands and the transformation is also shown in the Appendix. Composition in the context of data base operations is defined in the same way as with functions. An update command like

```
range of t is T
REPLACE t (Target-list) where Qualification
```

can be thought as the operation

$$T \leftarrow \mathbf{f}\,(T,R_1,R_2,\ldots,R_n)$$

where

$$\mathbf{f}\,(T,R_1,R_2,\ldots,R_n) = \begin{cases} \mathbf{h}\,(T,R_1,R_2,\ldots,R_n) & \text{if Qualification=true} \\ T & \text{if Qualification=false} \end{cases}$$

Here, **h** is a function that describes how values are assigned to fields of the relation T according to the Target-list.

The importance of this transformation relies on the fact that the relation updated is opened and accessed only once and moreover the query processing engine can make the new command more efficient than the separate execution of the two initial commands. The other advantage is in changing a loop to a single REPLACE loop and is shown with an example in section 4.

Examining other combinations of update commands we can see that there is no easy (and sometimes there is not at all a) way to combine two different commands in one. For example, an APPEND followed by a REPLACE cannot be generally changed to a single APPEND or REPLACE. It is also the case that two APPEND's or two DELETE's cannot be combined because the relation changes considerably by the addition or deletion of tuples, even in aggregate free commands.

3.3. Physical Database Design Techniques

Physical database design ([SCHO78]), assumes a given set of data, a set of commands to that data, and frequencies of those commands. It then derives a physical organization, or reorganization, of that data which will optimize the cost of the given set of commands. QUEL* optimization presents a similar problem: the optimizer is given a set of data, namely the given relations and their organization, plus a set of commands, and some information about the frequency of the commands. The optimizer seeks an optimal reorganization (perhaps none) of the physical database. What is missing is complete data on the frequency of the commands. For example, the QUEL* command:

```
REPLACE* emp (mgr = "Joe")
          where emp.mgr = emp1.name
          and    emp1.mgr = "Joe"
```

may benefit from the creation of indexes (if not already present) on the attributes "mgr" and "name". However, the benefit will depend on the number of times the REPLACE is executed. The optimizer must estimate the frequency of execution of each * command, using a heuristic such as those suggested in [KUNG84], where the same issue is discussed in the context of algorithm selection. With such frequency estimates, all the techniques of physical database design are potentially usable by the QUEL* optimizer. However, the QUEL* optimizer must take care not to reorganize the database in a way which will degrade future performance, e.g. creating an index which will slow down updates for future commands which do not use that index.

4. Transforming a QUEL* Command into a Single REPLACE

In this section we present two new optimization techniques which extend the technique of combining operations mentioned above. Each transforms a sequence of QUEL* commands into a single REPLACE command.

The transformation of several commands into a single REPLACE* command can yield significant savings. It allows the optimizer to concentrate its efforts on processing one canonical type of command, namely REPLACE. Since REPLACE does not change the size of the relation, the optimizer need make no estimates about that size. Processing need not involve the overhead of handling several different types of operators. Experimental evidence indicates that such a transformation does in fact save significant processing time for a particular class of problems.

4.1. Bounded Problem Space Problems

Consider a QUEL* command where only one relation, say R, is modified and this relation is known to be a subset of some other relation S, where S is known in advance of execution of the given QUEL* command. It is also known that R remains a subset of S throughout the execution of the given command. We will show that in this case it is possible to transform the given command to a single REPLACE or, in the case of EXEC*, in a single REPLACE* command.

In order to show that this transformation into a single command is possible, we first note the result of the appendix, which shows that any two REPLACE commands can be combined into a single REPLACE command. Thus we need only show that any data base operation on R can be expressed as a REPLACE command. We do that by constructing a relation S' which is equal to S with the addition of a new field, Present, with the following semantics :

a tuple from S that is currently in R will have a 1 in its Present field in S'

a tuple from S that is not currently in R will have a 0 in its Present field in S'.

We will now show that every database operation on R is equivalent to a REPLACE command on S'

— An APPEND command is transformed to a REPLACE command where the tuples that satisfy the qualification change their Present field value to 1. That is

```
range of r1,r2, ··· ,rm  is R
(range definitions for other tuple variables)

APPEND to R (f1 = val1, ...,fk = valk)
        where  w1 (r1,r2,...,rm,...)
```

becomes

```
REPLACE  s (Present = 1)
        where  s.f1  = val1
        and    s.f2  = val2
        and    s.fk  = valk
        and    w1 (s1,s2,...,sm,...)
        and    s1.Present=1
        and         .....
        and    sm.Present=1
```

— A REPLACE command remains exactly the same with the addition in the qualification of the clauses

```
        and    s1.Present=1
        and         .....
        and    sm.Present=1
```

for all tuple variables $r_1,r_2,...,r_m$ that range over R.

— A DELETE command is transformed to a REPLACE command where the tuples that satisfy the qualification change their Present field value to 0.

```
range of $r, r_1, r_2, \cdots, r_m$ is R
(range definitions for other tuple variables)

DELETE r
        where  $w_2(r_1, r_2, ..., r_m, ...)$
```

becomes

```
REPLACE  s (Present = 0)
        where   $w_2(s_1, s_2, ..., s_m, ...)$
        and     $s_1$.Present=1
        and        .....
        and     $s_m$.Present=1
```

In all of the above cases the tuple variables s_i range over S' and the qualification changes by adding clauses of the form

s_i.Present = 1

for all tuple variables r_i that range over the given relation R. This clause simply states that the tuples that should be referenced from S are only those that would normally be in R, i.e. those that result from APPEND or REPLACE commands (Present=1) and not those that have been deleted (Present=0). It is also clear that in the case of an APPEND command one need not include all fields in the new qualification. Only those fields that constitute a key should be included. The number of those is in most of the cases less than the total number of fields and the size of S much less than the cartesian product of the domains of the fields of R.

The reason for using the above transformation is that having a program with only REPLACE commands we can, using a systematic procedure, turn it to a single REPLACE* command (see Appendix). The problem is that in some cases the relation S is not known in advance or it is an extremely large relation. In the first case this transformation simply cannot be used and other optimization techniques must be used to get a better version for the program. In the second case it may still be possible to do the transformation. We shall see an example of this in the next section.

4.2. Dynamic Programming Problems

The problems we discuss in this section share the property that all are some implementation of the dynamic programming approach, in that a STATES relation, which contains (in each tuple) the current best value of the cost to be maximized, is built using the usual dynamic programming method. The example we use is a shortest path problem. The same tactic that we present here can be used with other standard applications of dynamic programming, e.g. the knapsack problem or the reliability problem.

A complete QUEL* program for our shortest path example appears in section 2.1 above. There the relation FEASIBLE is fixed and the relation STATES contains at all times the current state of knowledge about the problem. If we were to try to apply the technique of the previous section, we would seek a fixed relation S which contains STATES for the lifetime of the algorithm's execution. The problem is that such an S would have to hold a large number of tuples for each node, namely one tuple for every number less than the current cheapest cost of getting to that node. We will propose here a way to overcome this problem. The shortest path program, like any program using the dynamic programming approach, consists of two phases.

> In the first phase the relation STATES is expanded with the introduction of new nodes, i.e. the ones that can now be reached in the search space *(expansion phase)*
>
> Then in the second phase nodes with the same "node" value are compared and all but one are deleted according to some criterion, e.g. the cost of getting from the initial node to that specific node *(optimality phase)*

The main loop of the program would be

```
range of $r, r', r_1, r_2, \cdots, r_m$ is STATES
(range definitions for other tuple variables)

/* expansion phase */
APPEND to STATES (node = $val_0$, $f_1$ = $val_1$, ..., $f_k$ = $val_k$)
        where $w_1$ ($r_1, r_2, ..., r_m, ...$)

/* optimality phase */
DELETE r
        where r.node = r'.node
        and   w(r,r',...)
```

Moreover the function w is such that w(r,r',...) and w(r',r,...) cannot be both true (antisymmetric relation of r and r'). This means that only one tuple with a specific value of r.node will remain in the STATES relation after the optimality phase.

Let us now show that the above program can be transformed to a single REPLACE command. First, we add a field Present to the STATES relation and call the new relation S. We assume that initially all tuples in the S relation have their Present field with value 0. As was explained in the previous section the APPEND command will set the value to 1 while the DELETE will reset it to 0. Then the first command of the above program will be transformed to

```
range of $s,s',s_1,s_2, \cdots ,s_m$ is STATES
(range definitions for other tuple variables)

REPLACE s ($f_1$ = $val_1$, ...,$f_k$ = $val_k$,Present = 1)
        where   s.node = $val_0$
        and     $w_1$ ($s_1,s_2,...,s_m,...$)
        and     $s_1$.Present=1
        and       .....
        and     $s_m$.Present=1
```

Note that we have used the fact that "node" is a key in order to identify the tuple from S to be updated.

An attempt to transform the second command using the transformations from the previous section would fail since in the S relation there cannot be two tuples with the same "node" value. So the second command should be translated as follows

> if the tuple appended from the previous command is the first one appended to S for that value of the "node" field (i.e. r.Present=0), then do the update,
>
> else do the update only if the new tuple would not be deleted by the second command, i.e. if w(s,(val_0,val_1,val_2,...,val_k),...) is true, which guarantees that this tuple will not be deleted by the DELETE command.

This interpretation allows us to omit the DELETE command by only enhancing the qualification of the REPLACE command that replaced the initial APPEND operation (see similarities with the example presented in section 3). The final one-REPLACE command program will be

```
REPLACE  s (f1 = val1, ...,fk = valk,Present = 1)
        where   s.node = val0
        and     w1 (s1,s2,...,sm,...)
        and     s1.Present=1
        and        .....
        and     sm.Present=1
        and     (s.Present = 0 or w(s,(val0,val1,...,valk),...))
```

We should also note here that the query shown above might now be ambiguous since there may be many values to be assigned to a single tuple corresponding to the case of appending many tuples with the same "node" field value. This is the general problem of ambiguous updates and in our case is easy to solve by using the relation w to eliminate tuples. However, this will require the computation of the transitive closure of w over the candidate tuples.

We have shown how the above dynamic programming problem for search spaces has been reduced in a single REPLACE* program. The difference between the two programs is that the first one starts with a rather small relation which is incrementally growing as the iterations are executed while the second one starts with the whole problem space and updates the information about the nodes. What remains to be examined is how this new version compares in execution time and I/O operations with the initial version of the problem. The result of this comparison depends not only on the size of the S relation but also on the fraction of it that will be used in the program. It has been shown through a series of experiments that Dynamic Programming problems is a class of problems that will gain in execution time from this transformation ([KUNG84]).

5. Conclusion

We have described the problem of optimizing extended query language commands and in particular sequences of QUEL commands. We have presented several optimization tactics, some based on similar tactics in other areas and some new tactics. Our new tactics include the somewhat surprising result that any QUEL program satisfying certain criteria is equivalent to a QUEL program which consists of one REPLACE statement. We also show that a large class of problems, namely those which use the dynamic programming approach, satisfy these criteria. The transformations presented are useful not only in this context but in general transaction processing as well, since they are motivated solely by the need to expand the optimization unit form one database language command to a sequence of commands. The optimization techniques presented can be applied in a preprocessing phase, i.e. given a set of applications and the corresponding sequences of data base commands that implement them, one can apply our techniques in order to extract more information about the application and therefore design a more efficient execution pattern.

This is another level in the optimization hierarchy. Depending on the flexibility a language gives to the user, more or less optimization is possible; in data base languages especially where the expressive power of the language is much more restrictive than a general programming language, preprocessing can result in a much more efficient implementation. Normally the amount of information needed for the transformations we have described is minimal and the performance gains substantial. We will be able to verify these results by coding the transformations and measuring the performance of the two versions of various applications.

The extended query languages we have studied have been designed for engineering and for heuristic search applications. We hope in our future work to investigate the usefulness of our strategies especially in rule based systems and more generally production systems ([FORG79]). The existence of sets of rules which are thought of as parallelly executed commands, brings up the problem of efficient processing and execution. Deductive data bases ([WONG84]) is also another area where these optimization techniques can be used in order to implement efficiently the processing of inferences which can be thought of (for static environments) as predefined sequences of commands on the data base. In summary, we think that there is more mileage that can be gained in data base query optimization, especially as new constructs and extensions are proposed to support the need for more flexibility.

6. References

[AHO79] Aho, A., Ullman, J., "*Principles of Compiler Design*", Addison Wesley Co., 1979.

[AHO79b] Aho, A., Ullman, J., "*Universality of Data Retrieval Languages*", 6th ACM Symposium on Principles of Programming Languages, San-Antonio, TX, January 1979.

[BLAS76] Blasgen, M., Eswaran, K., "*On the evaluation of Queries in a Relational Data Base System*", IBM San Jose Research Report RJ-1745, April 1976.

[CHAM76] Chamberlin, D., et al., "*SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control*", IBM Journal Res.&Dev., Vol.20, No.6, Nov. 1976.

[EPST79] Epstein, R., "*Techniques for Processing Aggregates in Relational Database Systems*", UC Berkeley, Memo No. UCB/ERL/ M79/8 ,1979.

[FINK82] Finkelstein, S., "*Common Expression Analysis in Database Applications*", Proceedings of 1982 ACM-SIGMOD Conference on Management of Data, Orlando, FLA., June 1982.

[FORG79] Forgy, C., "*On the Efficient Implementation of Production Systems*", PhD Thesis, Carnegie-Mellon Univ., Pittsburgh, PA., 1977.

[GUTT84] Guttman, A., "*New Features for Relational Database Systems to Support CAD Applications*", PhD Thesis, University of California, Berkeley, June 1984.

[KACH82] Kachigan, S., "*Mulitvariate Statistical Analysis*", Radius Press, New York, 1982.

[KUNG84] Kung, R. et. al., "*Heuristic Search in Data Base Systems*", Proceedings of the 1st International Conference on Expert Data Base Systems, Kiawah Isl., SC., October 1984.

[PAPA79] Papadimitriou, Ch., "*The serializability of Concurrent Database Updates*", Journal of ACM, Vol. 26, No. 4, October 1979.

[SCHO78] Scholnick, M., "*A Survey of Physical Database Design Techniques*", Proceedings of the International Conference on Very Large Data Bases, 1978.

[SELI79] Selinger, P., et. al., "*Access Path Selection in a Relational Data Base System*", Proceedings of the 1979 ACM-SIGMOD Conference on the Management of Data, Boston, MA, June 1979.

[SELL82] Sellis, T., Ioannidis, Y.,"*Greek Language Phoneme Recognition With a Cluster Specification Method*", Undergraduate Thesis, National Technical University of Athens, Greece, June 1982 (in Greek).

[STON75] Stonebraker, M.,"*Implementation of Integrity Constraints and Views by Query Modification*", Proceedings of the 1975 ACM-SIGMOD Conference on the Management of Data, San Jose, CA, June 1975.

[STON76] Stonebraker, M. et. al., "*The Design and Implementation of INGRES*", ACM Transactions on Database Systems, September 1976.

[TSUR84] Tsur, S., Zaniolo, C., "*An Implementation of GEM - Supporting a Semantic Data Model on a Relational Back End*", Proceedings of the 1984 ACM-SIGMOD Conference on the Management of Data, Boston, MA, June, 1984.

[WONG76] Wong, E., and K. Youssefi, "*Decomposition: A Strategy for Query Processing*", ACM Transactions on Database Systems, September 1976.

[WONG84] Wong, E., et al., "*Enhancing INGRES with Deductive Power*", Position Paper, Proceedings of the 1st International Conference on Expert Data Base Systems, Kiawah Isl., SC., October 1984.

[ZANI83] Zaniolo, C., "*The Database Language GEM*", Proceedings of the 1983 ACM-SIGMOD Conference on the Management of Data, San Jose, CA, May 1983.

APPENDIX

The mechanical transformation proposed for changing two REPLACE's into a single REPLACE, is the following

Given

```
    Relation T ($f_1,f_2,\ldots,f_k$)

    range of $t,t_1,t_2,\ldots,t_n$ is T
    range of $s_1,\ldots,s_m,r_1,\ldots,r_l$ are other relations

(I) REPLACE t ($f_1=F_1(t,t_1,t_2,\ldots,t_n,s_1,s_2,\ldots,s_m)$,
              $f_2=F_2(t,t_1,t_2,\ldots,t_n,s_1,s_2,\ldots,s_m)$,
              ----  ----   ,
              $f_k=F_k(t,t_1,t_2,\ldots,t_n,s_1,s_2,\ldots,s_m)$)

    where QUAL1 ($t,t_1,t_2,\ldots,t_n,s_1,\ldots,s_m$)

(II) REPLACE t ($f_1=G_1(t,t_1,t_2,\ldots,t_n,r_1,r_2,\ldots,r_l)$,
               $f_2=G_2(t,t_1,t_2,\ldots,t_n,r_1,r_2,\ldots,r_l)$,
               ----  ----   ,
               $f_k=G_k(t,t_1,t_2,\ldots,t_n,r_1,r_2,\ldots,r_l)$)

    where QUAL2 ($t,t_1,t_2,\ldots,t_n,r_1,\ldots,r_l$)
```

transform it in the following query

```
    REPLACE t ($f_1=f'_1+(f''_1-f'_1)*d_2$.value,
               $f_2=f'_2+(f''_2-f'_2)*d_2$.value,
               ----  ----   ,
               $f_k=f'_k+(f''_k-f'_k)*d_2$.value)

    where
         [ (QUAL1 ($t,t_1,t_2,\ldots,t_n,s_1,\ldots,s_m$) and $d_1$.value=1)
           or
           (not QUAL1 ($t,t_1,t_2,\ldots,t_n,s_1,\ldots,s_m$) and $d_1$.value=0)
         ]
     and
         [ (QUAL2 ($t',t'_1,\ldots,t'_n,r_1,\ldots,r_l$) and $d_2$.value=1)
           or
           (not QUAL2 ($t',t'_1,\ldots,t'_n,r_1,\ldots,r_l$) and $d_2$.value=0)
         ]
```

where

$f'_i = t.f_i + [\, F_i(t,t_1,\ldots,t_n,s_1,\ldots,s_m) - t.f_i \,] * d_1.\text{value}$

$f''_i = G_i(t',t'_1,\ldots,t'_n,s_1,\ldots,s_m)$

t'_j = the tuple t_j where its fields $f_i (1 \leqslant i \leqslant k)$ are changed to f'_i

d_1 and d_2 are range variables over some dummy relation with a single field "value" and values 0 and 1 (a way to implement free variables)

Notice that this transformation can be done automatically. What remains to be seen is if the time of executing the two REPLACE commands separately is greater than the time of executing the new REPLACE command. Also note that we have generally expressed differences using the standard "-" operator. It is not difficult to define this operator for strings too so that if s is a string

$s-s=0$
$s+0=s$
$s-0=s$
$s*1=s$
$s*0=0$