A TRANSFORMATION-BASED APPROACH TO OPTIMIZING LOOPS IN DATABASE PROGRAMMING LANGUAGES

by

Daniel F. Lieuwen
David J. DeWitt

Computer Sciences Technical Report #1060

December 1991

A Transformation-based Approach to Optimizing Loops in Database Programming Languages

Daniel F. Lieuwen
David J. DeWitt

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

Database programming languages like O_2, E, and $O{+}{+}$ include the ability to iterate through a set. Nested iterators can be used to express joins. This paper describes compile-time optimizations similar to relational transformations like join reordering for such programming constructs. This paper also shows how to use a standard transformation-based optimizer to optimize these joins. An optimizer built using the EXODUS Optimizer Generator [GRAE87] was added to the Bell Labs $O{+}{+}$ [AGRA89] compiler. We used the resulting optimizing compiler to experimentally validate the ideas in this paper. The experiments show that this technique can significantly improve the performance of database programming languages.

The bulk of the implementation effort discussed in this paper was done by the first author at Bell Labs-Murray Hill. This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788 and by a donation from Texas Instruments.

1. Introduction

Many researchers believe that an object-oriented database system (OODBS) must be computationally complete—that programmers and database administrators must have access to a programming language to write methods and application programs [ATKI89]. While the programming language for such a system must include the ability to iterate through a set, giving programmers this power allows them to write programs that can be orders of magnitude slower than the desired computation should be. In order to solve this problem, compilers must be extended to include database-style optimizations.

It is especially important that compilers optimize set iterations that correspond to joins. There are at least three sources of such set iterations. First, it is unlikely that there will be a pointer between every pair of objects that are related in some fashion. Some relationships are needed infrequently, and thus are not worth storing explicitly; other relationships may be missed while designing the database. Thus, value-based joins will be needed, and they will sometimes be produced using nested iterators. Second, following pointers within a set iteration leads to an implicit join. If the system blindly follows pointers in the order specified by the user, the execution of the join may be unnecessarily slow [SHEK90]. Third, some joins will be produced by calling a function from within a set iteration—since the function may also iterate through a set. This paper will concentrate on value-based joins.

Since database programming languages such as PASCAL/R [SCHM77], O_2 [LECL89], E [RICH89], and O++ [AGRA89] provide constructs to iterate through a set in some unspecified order, it is possible to nest iterators in order to express value-based joins. The following is an example of a nested iterator expressed in O++:

```
(1) for (D of Division) {
        divisioncnt++;
        for (E of Employee) suchthat (E->division==D) {
            D->print();
            E->print();
            newline();
        }
    } /* a group-by loop */
```

We call the iteration through a set and its nested statements a **set loop**–the `for D` loop is a set loop that contains the statement `divisioncnt++` and another set loop. Due to the enclosed statements, the method of producing `Division|×|Employee` in (1) is more constrained than it would be in the relational setting–the join stream must be grouped by `Division`. We call loops like (1), where set loops contain other set loops (and possibly other statements), **group-by loops**. If each set loop, except the innermost, contains another set loop and no other statements, we say the loop is a **simple group-by loop**. If the statement `divisioncnt++` was removed from (1), query (1) would be a simple group-by loop. We call variables like `D` and `E` **iterator variables**.

As illustrated by (1), nested iterators can be used to express a join with a grouping constraint. However, as demonstrated in relational optimization, the associativity and commutativity properties of the join operator are vital properties during the optimization of a query. Thus, it is useful to remove as many ordering constraints as possible so that the join computation can be reordered, and, hence, the execution time of the program reduced. However, the flow of values through the program and the presence of output statements constrain the reorderings that can be made without violating the program's semantics. In this paper, we consider transformations that add extra set scans, use temporary sets, sort sets, and rewrite statements embedded in set loops to enable more reorderings to be made without modifying the program's semantics. These transformations are expressed as source-to-source transformations and as tree rewrites.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 defines the class of **self-commutative** statements. If a simple group-by loop contains a self-commutative statement, it can be optimized like a relational join. Section 4 describes the tree representation of a generic group-by loop. Section 5 uses the concept of self-commutativity and some analysis of the flow of values through the program to rewrite both simple group-by loops and more complicated group-by loops into a more efficient form. Each transformation is given in source-to-source form. Some representative transformations are also described as tree rewrites. Section 6 describes the implementation of these ideas in an optimizing compiler. Section 7 presents the results of experiments that demonstrate that this optimization technique can be quite useful. Our conclusions are contained in Section 8.

2. Related Work

[LIEU91] presents the transformations contained in this paper in source-to-source form and analytically evaluates the amount of I/O performed by the original and the transformed program assuming that all joins are computed using the hybrid hash join algorithm. This paper extends our earlier work in two ways. First, it shows how a standard transformation-based optimizer can be used to implement our program transformations. Second, this paper describes our implementation of an optimizer that employs our transformations for the Bell Labs *O++* compiler. The resulting optimizing compiler is used to demonstrate empirically that using these transformations can significantly improve performance for many programs.

[SHOP80] contains a slightly less-general version of the loop transformation called (T5) in this paper. [RIES83] uses an algebraic framework to optimize loops in the ADAPLEX database programming language. The algebra handles looping constructs more complicated than those covered in this paper. However, this algebra does not allow breaking a group-by loop into several loops, a technique used extensively in this paper. This algebra also does not recognize that certain nested loops are semantically equivalent to joins, an observation our optimizer exploits.

Our work is similar to work done in [KATZ82,DEMO85] to decompile CODASYL DML into embedded relational queries. Data flow analysis and pattern matching are used to transform CODASYL DML statements into DAPLEX-like statements. This transformation makes some flow of control statements unnecessary, so these statements are removed. Finally, the DAPLEX-like statements are transformed into relational queries. Both our work and theirs tries to take an imperative program and make it as declarative as possible while maintaining the semantics of the original program. Both use dataflow analysis and pattern matching. However, their work has a different objective than ours; their goal is to identify set loops in CODASYL DML and rewrite them as embedded relational queries. In our setting, the program syntax makes identifying set loops trivial. Our aim is to transform a group-by loop from the programmer specified form to a more efficient form. A key difference is that [KATZ82,DEMO85] ignored some semantic issues that are central to this paper. They only looked at DML statements and a few other COBOL commands that affect the flow of currency. Thus, they ignore grouping constraints, assuming that they exist only because COBOL DML cannot express a join without grouping constraints. This is reasonable, but it does require that the programmer check the transformed program to see if it has the proper semantics. Our use of the concept of self-commutativity allows us to convert group-by loops into joins without modifying a program's semantics. Their work (once modified to take grouping constraints into account) can be used as a preprocessing step that allows our transformations to be applied.

The work in this paper is also related to work on transforming nested query blocks in SQL into equivalent queries with no nesting. [KIM82,DAYA87,GANS87,MURA89] start with a simple kind of nested query and show how to transform it into a join query without a nested query in the **where** clause. Other transformations take a more complicated nested query and produce two or more subqueries that compute the same result. Some subqueries are not flat, but their nesting patterns are simpler than the nesting pattern of the original query. These subqueries can be simplified further by other transformations. We, too, break a complicated subquery into several parts. We then further transform the resulting subqueries just as they do.

The idea of interchanging loops appears frequently in work on vectorizing FORTRAN [PADU86,WOLF86,WOLF89]. For instance,

```
(2) do I = 1, N
       do J = 1, N
           S = S + B(I,J)
           A(I,J+1) = A(I,J)*B(I,J) + C(I,J)
       enddo
    enddo
```

cannot be directly vectorized. However, if we interchange the `I` and `J` loops, the definition of `A(I,J+1)` can be vectorized. The definition of `S` involves a reduction operation. A reduction operation reduces the contents of an array to a single quantity in an order independent manner—examples include producing the sum or product of array

elements. Reductions do not inhibit loop interchange if the user allows loop interchange to be carried out (because of the finite precision of computer arithmetic, interchanging loops for a reduction can lead to a different answer even though mathematically the same answer should be computed). The interchanges are only performed if loop-carried dependences satisfy certain properties. We also use dataflow analysis to interchange loops. However, since our emphasis is on sets and not arrays, our analysis has a different flavor. Thus the general idea is similar although the analysis used is different.

Loop fission has been used to optimize FORTRAN programs. Loop fission breaks a single loop into several smaller loops to improve the locality of data reference. This can dramatically improve paging performance [ABU81]. Our transformations serve a similar function–breaking a large loop into several small ones to enable database-style optimization.

3. Introduction to Self-commutativity

Before examining the different transformation strategies that we have developed, we first examine when a simple group-by loop can be optimized like a relational join, since this is the base case of our optimization strategy. We introduce *O++* [AGRA89] syntax for expressing a join[1]. The SQL query

(3) **select**(D.all, P.all) **from** D **in** Dept, P **in** Professor **where** D.did=P.did

can be expressed in *O++* without adding unnecessary constraints as the following **join loop**:

```
(4) for (D of Dept; P of Professor) suchthat (D->did==P->did) {
        D->print();
        P->print();
        newline();
    } /* a join loop */
```

Ignoring output formatting, the two statements are equivalent. To identify when a simple group-by loop can be rewritten as a join loop, we introduce the concept of a statement being **self-commutative** relative to a set of nested loops. Consider the following simple group-by loop:

```
(5)  for (X1 of Set1) suchthat (Pred1(X1))
         ...
         for (Xm of Setm) suchthat (Predm(X1, ..., Xm))
             S;
```

Since the order of iteration through each of the sets Set_1, Set_2, ..., and Set_m is unspecified, after (5) has been executed, there is a set of alternative program states that may be reached (a program's state is determined by the values of variables, the output produced, etc...). Since the actual state reached may vary from program run to program

[1]The syntax presented here is different than that presented in [AGRA89]. It is the syntax of the present *O++* compiler.

run, programs containing statements like (5) are potentially non-deterministic. The **meaning** of (5) in a particular starting program state is the set of states that may be reached from that starting state by executing (5).

Definition: The statement S in (5) is **self-commutative relative to** X_1, X_2, ..., **and** X_m if the meaning of

```
(6) for (X1 of Set1; ... ; Xm of Setm) suchthat
    (Pred1(X1) && ... && Predm(X1, ..., Xm))
          S;
```

in any starting program state is identical to the meaning of (5) in the same starting state and this meaning is a singleton set (i.e. (5) and (6) produce an identical, deterministic result).

We will leave off the phrase **relative to** X_1, X_2, ..., **and** X_m unless it is necessary for clarity. This definition requires that the inner/outer relationship among the sets can be permuted arbitrarily during the evaluation of the join and that any join method can be used without changing the final computation of the program. It should be noted that this definition is only satisfiable if none of the sets are physically or logically embedded in other sets (i.e. $\nexists\, i<j$ s.t. $Set_j = X_i$->set). If such embeddings exist, the inner/outer relationships among sets must obey the partial ordering that if a set Set_{in} is embedded in an object of set Set_{out}, then the join must have the set loop for Set_{in} inside the set loop for Set_{out}.[2]

In this paper, we assume that computer arithmetic is associative and commutative. Given this assumption, the computation of a sequence of aggregates is a self-commutative statement. For example, in

```
(7) for (D of Division)
        for (E of Employee) suchthat (E->division==D) {
            totpay += (E->basepay*D->profitshare)/100 + ChristmasBonus;
            empcnt++;
        }
```

the statement sequence that increments totpay and empcnt is self-commutative. Since integer addition is assumed to be associative and commutative, an arbitrary pair of instantiations of the following two program statements

```
(8)    totpay += (E->basepay*D->profitshare)/100 + ChristmasBonus;
       empcnt++;
```

such as

[2] If there is another set that contains all the objects of an embedded set (for instance, if there is an extent, a set of all the objects of a particular type), (5) could be rewritten to access them through this other set. The rewritten query might well have no embedded sets. Then S might be self-commutative relative to the new list of sets even though it was not **self-commutative relative to** X_1, X_2, ..., **and** X_m. We will ignore such rewrites in this paper; [SHEK90] employs this technique.

```
(9)    totpay += (20000*110)/100 + 500; empcnt++;
       totpay += (30000*120)/100 + 500; empcnt++;
```

can be flipped without changing the final value of `empcnt` or `totpay`. Statements like (8) are termed **reductions** because they reduce a subset of a set or Cartesian product to a single value in an order independent manner. Reductions are self-commutative.[3] A more complete description of the class of self-commutative statements is presented in [LIEU91].

4. Introduction to the Representation

The compiler's abstract syntax tree (AST) could be used as the query representation during the optimization of a group-by loop. The transformations contained in this paper in source-to-source form have straightforward analogues in AST-to-AST form. A typical AST representation of

```
for (D of Division) {
     divisioncnt++;
     for (E of Employee) suchthat (E->division==D) {
         D->print();
         E->print();
         newline();
     }
 } /* group-by loop */
```

has a node representing the `for D` loop. One of the attributes of this node is a list of statements directly contained in the `for D` loop—`divisioncnt++` and the `for E` loop. While this is a good representation for generating code from, it is a poor one for database-style optimization optimization. We wished to use an optimizer generator [GRAE87,LOHM88], a tool that take a set of rules and produces an optimizer. However, the EXODUS Optimizer Generator, the only generator we had access to, expects all operators to have fixed arity. If we use an AST representation, a set loop is an operator with as many operands as there are statements directly contained in the set loop. Since we wished to avoid designing a search strategy, this made an AST representation unacceptable. Even if we had been willing to build the whole optimizer by hand, this representation would have been poor, because the optimizer would need to regularly search lists of statements to see if they contain set loops. This could make the optimization process much slower. In short, an AST representation doesn't make set loops sufficiently prominent, which is a major limitation since set loops are the most important constructs for database-style optimization.

To avoid these problems, we developed our own tree representation of set loops. We transform the AST into our new representation as the first step in the optimization process. Consider the following generic template for a query:

---

[3]We are using FORTRAN optimization terminology. An APL/LISP reduction is not necessarily an order independent operation.

```
(10) for (X1 of Set1) suchthat (Pred1,1(X1)) {
        S1,1;
        ...
        for (Xi of Seti)  suchthat  (Predi,1(Xi) && Predi,2(X1,...,Xi)){
            Si,1;
            ...
            for (Xn of Setn)  suchthat  (Predn,1(Xn) && Predn,2(X1,...,Xn)) {
                Sn,1;
            }
            ...
            Si,2;
        }
        ...
        S1,2;
    }
```

In (10), $Pred_{i,1}(X_i)$ is a predicate that only involves X_i, constants, and variables constant for the duration of (10). $Pred_{i,2}(X_1,...,X_i)$ contains all clauses of the predicate belonging to the **for** X_i loop that are not contained in $Pred_{i,1}(X_i)$. (The original query may not have the predicates in this form, and so the system may have to manipulate them into this form.) We represent the query (10) as the tree in Figure 1.

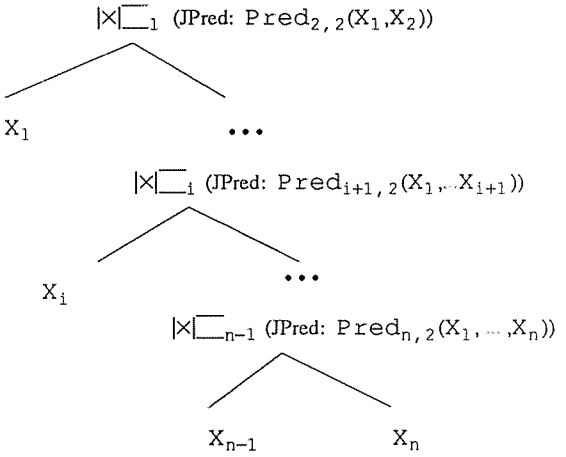

Figure 1: Tree Representation of (10)

Var = X_i
First = $S_{i,1}$
Second = $S_{i,2}$
StmtsDesc = <*do data flow analysis*>
Flow = <*do data flow analysis*>
SetName = Set_i
SelectPred = $Pred_{i,1}(X_i)$

Figure 2: Contents of the X_i **Set Node**

The transformation process is mechanical. Each leaf node represents a set iteration. For example, the left-most node in Figure 1 corresponds to the **for** X_1 set loop in (10). We will identify a leaf node by the iterator variable of the corresponding set loop (e.g. the node corresponding to the **for** X_1 set loop in (10) is called the X_1 node). Figure 2 describes the contents of the X_i set node. The *First* field represents the code in the **for** X_i set loop that precedes the embedded **for** X_{i+1} set loop (e.g. in the X_1 node, since $S_{1,1}$ precedes the **for** X_2 loop, *First* has the value $S_{1,1}$). The *Second* field represents the code that follows the enclosed set loop. If there is no enclosed set loop, *First* contains all the code enclosed in the node's set loop, and *Second* is *NULL*.

The *StmtsDesc* (statements' description) field contains some aggregate information about the code represented by *First* and *Second*. The *StmtsDesc* field has the value *self-comm* if the sequence $S=S_{i,1}; S_{i,2}$ would be self-commutative

relative to a simple group-by loop over Set_1-Set_i with S as the inner statement[4], *empty* if the sequence is empty, and *not-self-comm* otherwise. This labeling is useful for our purposes because a sequence of transformation applications may produce a simple group-by loop over the sets Set_1-Set_i (or over temporary sets produced by applying selections and projections to Set_1-Set_i) containing just S, $S_{i,1}$, or $S_{i,2}$ (or a version of the statement S, $S_{i,1}$, or $S_{i,2}$ modified to use the temporary sets instead of the original sets). If S is self-commutative in the sense above, the simple group-by loop subquery will contain a self-commutative statement.

The *Flow* field describes the flow of values: *noflow* means no values flow from the statements represented by *First* or *Second* to the enclosed X_{i+1} set loop or from the X_{i+1} set loop to the **for** X_i loop; *flow-in* means values flow from *First* or *Second* into the X_{i+1} set loop, but no values flow from the X_{i+1} set loop to *First* or *Second*; *aggregate* means that the the X_{i+1} loop calculates an aggregate (i.e. values flow from *First* to the X_{i+1} loop and from the X_{i+1} loop to *Second*, but in a constrained way); *innermost* means no set loop is contained in the X_i loop. If output is produced in *First* or *Second* and in the X_{i+1} loop, but there is no other flow of values from the X_{i+1} loop to *First* or *Second*, *Flow* is labeled *io-flow-in*. If none of these conditions hold, the *Flow* field will have the value *flow-out-from-inner*.

The $|\times|\overline{\ }_j$ $\forall j\ 1 \le j < n$ operator's left-hand child represents the **for** X_j loop and all the statements contained in it except the **for** X_{j+1} loop. The right-hand child represents the complete **for** X_{j+1} set loop, except for those parts of the predicate that refer to the iterator variables X_1-X_j or to variables modified in the group-by loop (10). Those parts of the predicate (i.e. $Pred_{j+1,2}(X_1, \ldots, X_{j+1})$) are assigned to the *JPred* field of the $|\times|\overline{\ }_j$ node.

The representation presented here is too simple for optimizing loops that modify elements of the sets being iterated over. Simple extensions could be made that mark the X_i set node if any of the elements of Set_i may be modified by the group-by loop. If so, the optimizations described below that involve projecting Set_i to produce a temporary set and then using the temporary set instead of the original set must be suppressed. In this case, we need access to the original objects; access to projected sub-objects in a temporary set will not suffice. Since this extension is straightforward, we leave out the necessary details to avoid cluttering the exposition. We assume that none of the predicates have side-effects. We also assume that the predicates do not use variables modified by program statements. This is not strictly necessary—predicates involving such variables can be handled, but handling them requires complicating the representation and the exposition.

---

[4]Making the *StmtsDesc* field single-valued does eliminate some optimization possibilities since a statement may be self-commutative relative to some, but not all surrounding loops. However, there are only three classes of statements in [LIEU91] where this is true. The first is a reduction operation on variable v where v is used in the predicate of a surrounding loop. Not only are we not covering this case in this paper, it also seems like a rare case. Thus being able to handle this case better is an insufficient reason to justify complicating the optimization process. The second is an insertion into a set being iterated through. This is a fixed point query, and so other techniques are more appropriate to handling it. The third is the deletion from a set being iterated over. However, it will almost always be the case that the deletion will be from the innermost set—in which case, the statement will not be self-commutative relative to any nesting of loops. Thus, this choice of representation appears to be the best choice; it is simple, and it allows the optimization of plausible programs.

The tree representation used here is different from a typical representation used to represent relational joins, the left deep query tree [GRAE87]. A tree's cost function cannot be evaluated bottom up unless each leaf node maintains information about how many times the set loop corresponding to the node is expected to be executed (e.g. For the X_1 node in (10) the value would be one. For the X_2 node, it would be the number of selected objects from Set_1 since the **for** X_2 loop will be executed once for each of them.) This makes it more difficult either to evaluate the cost functions or to apply the transformations. However, this representation has two major advantages. First, it is easier to represent the transformations contained in this paper via tree manipulation using this representation than would be the case using a more conventional tree representation of a join. Second, subqueries are represented naturally as subtrees, and a subtree can be optimized independently of the overall tree. In a left-deep query tree representation, no subtree corresponds to the **for** X_i loop in (10) $\forall i\ 1<i<n$.

The *First* and *Second* fields are not used during optimization. The fields *StmtsDesc* and *Flow* contain sufficient information for the optimization process. In fact, during optimization, the code sequences in the *First* and *Second* fields are not updated by our implementation. Instead, a log of transformations that would require the statements corresponding to *First* and *Second* to be rewritten is maintained. For the plan that is chosen by the optimizer, the transformations used to transform the initial plan to the final plan are found in the log. The appropriate transformations of the *First* and *Second* fields are then applied to produce the final query. Since the code sequences can be fairly long, this approach can potentially speed up optimization by eliminating useless work. However, in the examples below, we update the *First* and *Second* fields so that the correspondence between the textual and tree representations of the code is kept clear.

5. Transformations

In the transformations that follow, *Ta* and *Tb* will represent arbitrary trees, while *X* and *Y* will represent either set iteration nodes or Supernodes. A Supernode represents a nested set loop. Nested set loops with no intervening statements can be treated as a single set loop over an implicit set. For instance, in the query

```
(11) for (X of SetX) suchthat (P1(X))
          for (Y of SetY) suchthat (P2(Y) && P3(X,Y)) {
              S1;
              for (Z of SetZ) suchthat (P4(Z) && P5(X,Y,Z))
                  S2;
              S3;
          }
```

the two outer loops can be treated as a single loop to produce:

```
(12) for (X of SetX; Y of SetY) suchthat (P1(X) && P2(Y) && P3(X,Y))
     /*sort*/ by (X->key) {
         S1;
         for (Z of SetZ) suchthat (P4(Z) && P5(X,Y,Z))
             S2;
         S3;
     }
```

In (12), the join of `SetX` and `SetY` is produced and then sorted by the key of `SetX` (which may be the object identifier). The observation that allowed query (11) to be transformed to (12) can also be represented using Figure 3. We represent a |×|‾ operator with |×|• if the left-hand child's *StmtsDesc* is empty. The node with a *Var* value of *V* on the right-hand side of a transformation corresponds to the node labeled *V* on the left-hand side. In each of the tree transformations, the right-hand side will show only those fields of the *V* node that have changed from the left-hand side.

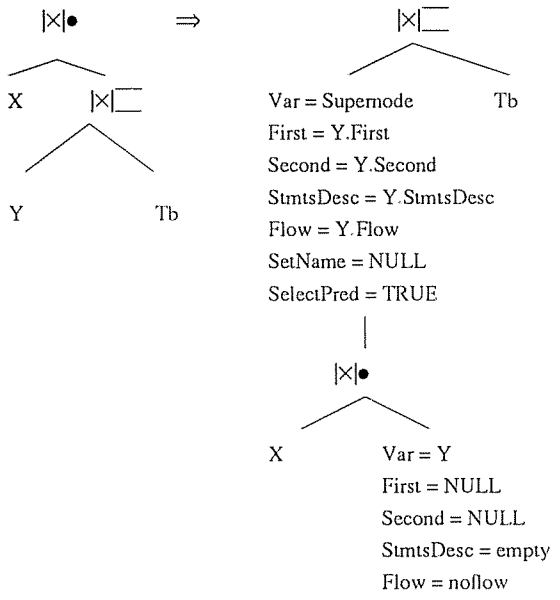

Figure 3: Supernoding

Since the operator on the left-hand side of Figure 3 is |×|•, the *First* and *Second* fields of the `X` node are NULL. This corresponds to the empty statements that precede and follow the **`for`** `Y` loop in (11). We move the *First*, *Second*, *StmtsDesc*, and *Flow* fields of the `Y` node up to the Supernode, which represents the observation that the **`for`** `X` and **`for`** `Y` set iterations can be treated as a single set loop over an implicit set. This implicit set has no name, and there is no selection predicate on it.

As an example of this transformation, consider:

```
(13) for (X1 of Set1) suchthat (Pred1,1(X1))
       for (X2 of Set2) suchthat (Pred2,1(X2) && Pred2,2(X1,X2))
          for (X3 of Set3) suchthat (Pred3,1(X3) && Pred3,2(X1,X2,X3)) {
            S1;
            for (X4 of Set4) suchthat (Pred4,1(X4) && Pred4,2(X1,...,X4))
               S2;
            S3;
          }
```

which can be represented using our tree notation as shown in Figure 4 (some unnecessary detail has been omitted). Applying the Supernoding transformation twice produces Figure 5 (the X_1-X_3 nodes have *First*=*Second*=NULL in Figure 5).

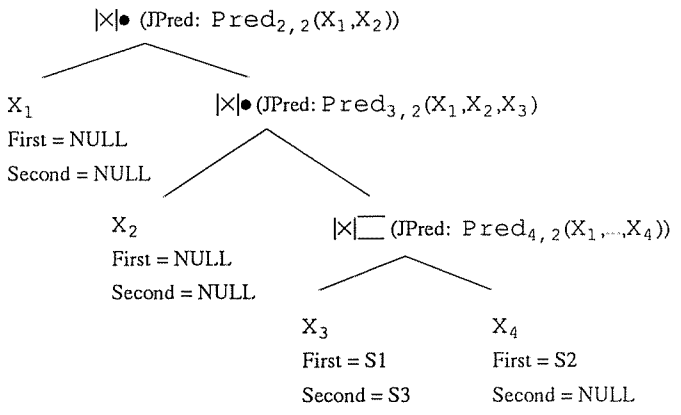

Figure 4: Tree Representation of (13)

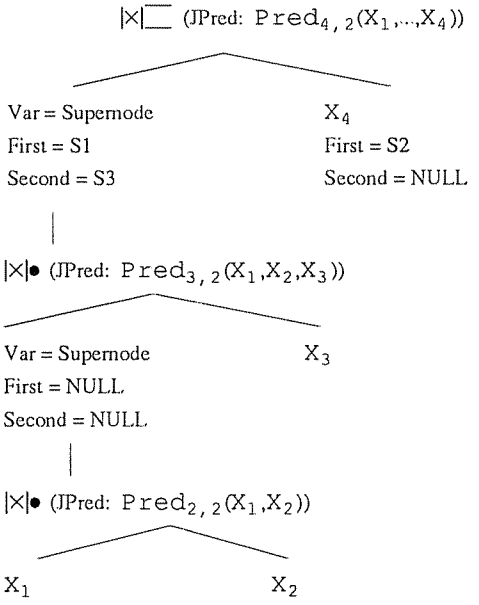

Figure 5: Tree Representation of (13) after two applications of Supernoding transformation

Note that the tree representation in Figure 5 would closely resemble a standard tree representation for relational joins, the left-deep query tree [GRAE87], if the Supernodes were removed from the tree.

5.1. Simple group-by loops

Having introduced our basic terminology and the tree representation that is used for unoptimized queries, we next consider a variety of transformations that we have developed. The simplest loops that may be rewritten are of the form:

```
(14) for (X1 of Set1) suchthat (Pred1(X1))
       ...
        for (Xm of Setm) suchthat (Predm(X1, ..., Xm))
            S;
```

If S is a self-commutative statement, then, by the definition of self-commutativity, (14) is equivalent to:

```
(T1) for (X1 of Set1; ... ; Xm of Setm) suchthat
     (Pred1(X1) && ... && Predm(X1, ..., Xm))
          S;
```

Even if S is not self-commutative, if S does not modify Set_1-Set_m or the variables used in the predicates, (14) can be rewritten as a join followed by a sort:

```
(T2) Temp = {};
     for (X1 of Set1; ... ; Xm of Setm) suchthat
     (Pred1(X1) && ... && Predm(X1, ..., Xm))
          Insert <Needed(X1), ..., Needed(Xm)> into Temp;
     Sort Temp on composite key (X1, X2, ..., Xm-1);
     for T in Temp /* in the sorted order */
          S';
```

In transformation (T2), the Temp set loop contains a statement S′ that looks like S, except that uses of the fields of Set_i $\forall i\ 1 \le i \le m$ are replaced by uses of the fields of Temp. $Needed(X_i)$ refers to the fields of Set_i that either are used in statement S or are needed for the sort. It includes a unique identifier for X_1-X_{m-1} for use in sorting; if the user has not supplied a primary key, $Needed(X_i)$ includes the object identifier (oid) of X_i as the identifier. The asymmetric treatment of X_m in transformation (T2) allows the transformed program to maintain the proper semantics while minimizing cost. Statement (14) has a non-deterministic execution order. Program semantics do not require iterating through the sets in the same order each time; they only require that X_1 varies the most slowly, followed by X_2, X_3, ... Sorting on the composite key maintains something slightly stronger than this semantic requirement. The transformed program will behave as if it was iterating through Set_1-Set_{m-1} in the same order each time, although it may behave as if it was iterating through Set_m in a different order each time. Thus, the asymmetric treatment maintains the program semantics without wasting space in each Temp object for a unique identifier for the relevant X_m object.

Since (T1) and (T2) are so similar, we will only show the tree transformation for (T2). (T2) can be represented as Figure 6. The tree form of transformation (T2) will apply if *Y* is a set node, and *Y.StmtsDesc* is not *empty* (i.e. X|×|•Y is not the child of a Supernode).

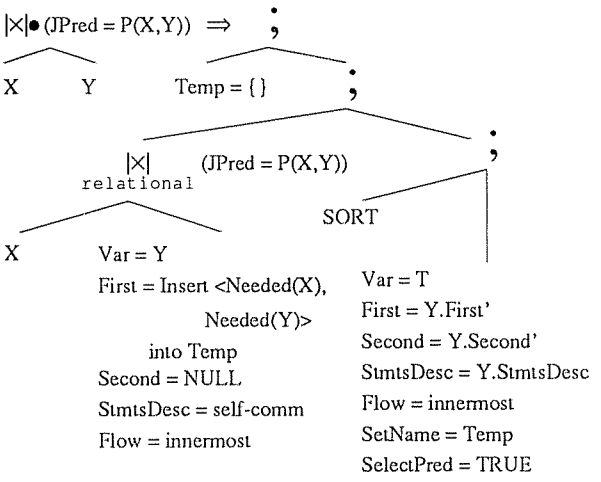

Figure 6: Transformation (T2)

The correspondence between (T2) and Figure 6 is straightforward. The ; operator represents sequencing–the actions of the left child are performed before those of the right child. Since the original operator on the left-hand side of the transformation is |×|•, we know that the *X* loop contains the *Y* loop and no other statements in both the original and the transformed version. Also, since `X` can be either a set node or a Supernode, it can represent either one or several sets; it corresponds to the `(m-1)` outer sets in (14). The $\underset{\text{relational}}{|\times|}$ subtree corresponds to the first set loop in (T2), the loop that inserts into `Temp`. The optimizer knows that the elements of the $X \underset{\text{relational}}{|\times|} Y$ join stream can be produced in any order. Remember that we have already made the tree on the left-hand side of Figure 6 closely resemble the left-deep query tree for a relational join by applying the Supernoding transformation. The optimizer can use slightly modified versions of standard relational transformations to optimize the $X \underset{\text{relational}}{|\times|} Y$ subquery. Since the standard relational transformations are well-known and have natural analogues here, we will not cover them in this paper. The `SORT` node in Figure 6 corresponds to the sort in (T2). The *T* set node corresponds to the final loop in (T2)–in both, we loop through the temporary set executing a modified version of the statement contained in the innermost loop.

5.2. General Group-by Loops

The group-by loops exemplified by (14) are the simplest possible–each set loop except the innermost contains a single statement, a set loop. Only the innermost loop contains a statement sequence. In general, however, each set loop will contain a statement sequence. In other words, the query

```
(15) for (X of SetX)  suchthat (Pred1(X)) {
         S1;
         for (Y of SetY)  suchthat (Pred2(Y) && Pred22(X,Y))
             S2;
         S3;
     }
```

exemplifies the general case. In the following sections, we will assume that `S1`, `S2`, and `S3` do not modify `SetX` or `SetY`.

5.2.1. General Group-by Loops Without Flow Dependencies into the Inner Loop

In (15), if a variable is defined in both `S2` (the inner loop) and in `S1` or `S3` (the outer loop) and both definitions reach a use outside the inner loop, we will say the two loops **interfere.** Suppose the loops do not interfere, and no values flow from the outer to the inner loop or from the inner to the outer loop, then (15) can be rewritten as (T3) provided that at least one of the following three conditions hold: (1) the sequence `S1;S3` is self-commutative relative to `X`; (2) `S2` is self-commutative relative to `X` and `Y`; or (3) `SetX` is iterated over in the same order in both set loops in (T3).

```
(T3) for (X of SetX)  suchthat (Pred1(X)) {
         S1;
         S3;
     }
     for (X of SetX))  suchthat (Pred1(X))
         for (Y of SetY)  suchthat (Pred2(Y) && Pred22(X,Y))
             S2;
```

Requiring that one of the three conditions mentioned above hold is necessary in order to avoid subtle forms of inconsistency. [LIEU91] contains an example of a violation of program semantics that can occur if transformation (T3) is applied when these conditions are not met.

As an example of a case where rewrite (T3) is profitable, consider counting the number of professors on each floor assuming that all the professors of a given department work on the same floor.

```
(16) for (D of Dept) {
         deptcnt++;
         for (P of Professor) suchthat (P->did==D->did)
             cnt[D->floor]++; //S2
     }
```

(T3) and a loop interchange can be applied to rewrite (16) as:

```
(17) for (D of Dept)
         deptcnt++;

     for (P of Professor)
         for (D of Dept) suchthat (P->did==D->did)
             cnt[D->floor]++; // Calculate number of professors/floor
```

If the Professor extent does not fit in the buffer pool, but the Dept extent does, (17) will be incur fewer I/Os than (16) no matter what join algorithm is used to evaluate (16). Query (17) will scan Dept once to load Dept objects into the buffer pool and calculate deptcnt. The Professor|×|Dept join will only require a single scan of the Professor extent since Dept is already in the buffer pool. Query (16), on the other hand, must reread part of the Professor extent whether it evaluates the join with a hybrid hash [DEWI84], a sort-merge, or a nested loops join algorithm. Thus, query (16) will incur more I/Os than (17).

5.2.2. General Group-by Loops With Flow Dependences into the Inner Loop

Transformation (T3) can be applied if no information flows in either direction between the inner and outer loop. In statements of the form:

```
for (X of SetX)  suchthat (Pred1(X)) {
    S1;
    for (Y of SetY)  suchthat (Pred2(Y) && Pred22(X,Y))
        S2;
    S3;
}
```

however, values computed in the outer loop will often be used in the inner. If no values flow from S2 to the outer loop, the loops do not interfere, and S2 does not modify any elements of SetX, but values do flow from S1 or S3 into the inner loop, a somewhat more complicated transformation than (T3) is required. Let $v_1, \ldots, v_n$ be the variables written by S1 and S3 that are used by expressions in the inner loop. Then this query can be rewritten as:

```
(T4) Temp = []; //empty sequence

     for (X of SetX) suchthat (Pred1(X)) {
         S1;
         Append <Needed(X), v1, ... ,vn> to Temp.
         S3;
     }
     for (T of Temp)  /* in insertion order */
         for (Y of SetY) suchthat (Pred2(Y) && Pred22(X,Y)')
             S2';
```

S2′ is S2 rewritten to use fields of Temp instead of v_i $\forall i\ 1 \le i \le n$ and instead of fields of SetX. Pred22(X,Y)′ is Pred22(X,Y) similarly rewritten. Needed(X) contains the fields of X used in Pred22(X,Y) and S2. Note that if Pred1(X) is very restrictive or objects in Temp are shorter than objects in SetX, (T4) will sometimes be preferable to (T3) because the cost of storing and rereading the necessary objects will be less than the cost of recomputing the stream. It should also be kept in mind that neither transformation (T3) nor (T4) will be useful unless the simple group-by loop produced by the transformation can be further transformed, S1 or S3 contains a statement that may cause disk activity (i.e. pointer dereferencing or a set loop), or multi-query optimization can be

used to eliminate the cost of the extra scan of SetX.

Since Temp is a sequence (i.e. an ordered set) (T4) iterates through Temp in insertion order. This ensures that both the original and the transformed program behave in the same manner. If, however, S2 is self-commutative, it is not required that (T4) iterate through Temp in any particular order, and so Temp may be a set.

Transformation (T4) can also be expressed using the tree shown in Figure 7.

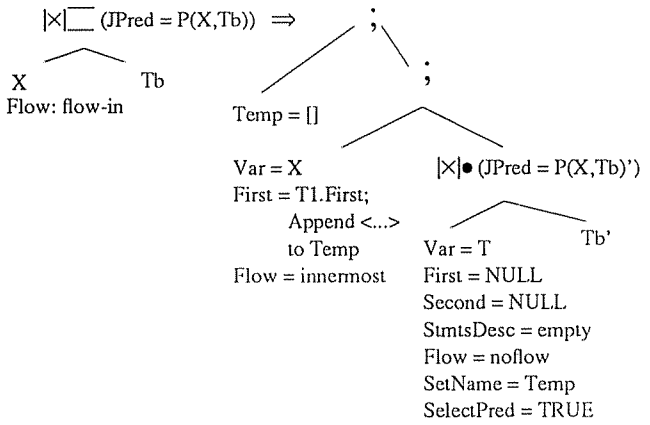

Figure 7: Transformation (T4)

Tb' means that the predicates and statements in tree *Tb* must be rewritten to use fields of Temp instead of v_i $\forall i\ 1 \le i \le n$ and instead of fields of the outer set(s) included in the *X* subtree. As an example of a case where rewrite (T4) is profitable, consider counting the number of professors on each floor assuming that all the professors of a given department work on the same floor. Suppose the floor of a Dept is not directly stored in the Dept object. Instead, it is contained in an object that contains information about the part of the building belonging to the Dept. Query (18) will then calculate the desired result.

```
(18) for (D of Dept) {
         floor = D->buildinginfo->floor; //S1
         for (P of Professor) suchthat (P->did==D->did)
             cnt[floor]++; //S2
     }
```

(T4) and a loop interchange can be applied to rewrite (18) as:

```
(19) Temp = {}; //S2 is self-commutative
     for (D of Dept) {
         floor = D->buildinginfo->floor;
         Append <D->did, floor> to Temp;
     }
     for (P of Professor)
         for (T of Temp) suchthat (P->did==T->did)
             cnt[T->floor]++; // Calculate number of emps/floor
```

If the Professor extent does not fit in main memory, but the Dept extent does, (19) will be incur fewer I/Os than (18) no matter what join algorithm is used to evaluate (18). Both queries require the same number of I/Os to calculate D->buildinginfo->floor. If Dept fits in main memory, so will Temp. Query (19) will scan Dept once to create Temp and load it into the buffer pool. The Professor|×|Temp loop will only require a single scan of Professor since Temp is already in the buffer pool. Thus, (19) requires only a single scan of the Dept and Professor extents. Query (18), like query (16), must reread part of the Professor extent no matter what join algorithm is used to evaluate the join. Thus, query (19) will incur fewer I/Os than (18).

5.2.3. General Group-by Loops Used As Aggregates On Grouped Values

The transformations presented so far do not allow the optimization of aggregate functions such as:

(20) **select**(D.name, count(*)) **from** D **of** Dept, P **of** Professor **where** D.did=P.did
group by D.name

This SQL query can be expressed in *O++* as:

```
(21) for (D of Dept) {
         cnt = 0;  //S1
         for (P of Professor) suchthat (D->did==P->did)
             cnt++;  //S2
         printf("%s %d", D->name, cnt); newline(); //S3
     }
```

We consider a transformation from [SHOP80] to rewrite queries involving aggregate functions such as (21). Consider

```
(22) for (X of SetX) suchthat (Pred1(X)) {
         S1;
         for (Y of SetY) suchthat (Pred2(Y) && Pred22(X,Y))
             S2;
         S3;
     }
```

Suppose that S1 can be partitioned into two sets of statements: those whose values flow only to S1 and to outside the **for** X loop and those that assign constants to variables v_1, ..., v_n. The v_1, ..., v_n must be assigned to during each pass through S1. In S2, they may only be employed in reduction operations; in S3, they may be read but not written. Statements in S3 may only have values flow back to S3 and to outside the **for** X loop. S2 and S3 must not modify any elements of SetX. Finally, SetY must not be nested inside an object of SetX. If these conditions are met, (22) can be rewritten as:

```
(T5) Temp = []; //empty sequence
     for (X of SetX) suchthat (Pred1(X)) {
         S1;
         Insert <Needed(X), v1, ..., vn> into Temp.
     }
     for (T of Temp; Y of SetY) suchthat (Pred2(Y) && Pred22(X,Y)')
         S2';

     for (T of Temp)
         S3';
```

provided `S2'` is self-commutative relative to `Y` and `T`. `Needed(X)` are the fields of `SetX` mentioned in `S2`, `S3`, and `Pred22(X,Y)`. `S2'`, `S3'`, and `Pred22(X,Y)'` are rewritten versions of `S2`, `S3`, and `Pred22(X,Y)` that replace uses and definitions of v_i with uses and definitions of `T->`v_i. They also replace uses of fields of `SetX` with uses of the fields of `Temp` (i.e. `X->a` is replaced with `T->a`). If `S1` or `S3` is self-commutative, `Temp` can be a set provided `SetX` is an explicit set and not the implicit set resulting from a join (like the outer loop in (12)).

Consider applying (T5) to the following loop:

```
(23) for (D of Dept) {
         cnt = 0;  //S1
         for (P of Professor) suchthat (D->did==P->did)
             cnt++;  //S2
         printf("%s %d", D->name, cnt); newline(); //S3
     }
```

Applying (T5) to (23) produces:

```
(24) Temp = [];
     for (D of Dept) {
         cnt = 0;  //S1
         Insert <D->did,D->name,cnt> into Temp;
     }
     for (P of Professor)
         for (T of Temp) suchthat (T->did==P->did)
             T->cnt++;

     for (T of Temp) {
         printf("%s %d", T->name, T->cnt); newline(); //S3
     }
```

If the `Professor` extent does not fit in main memory, but the `Dept` extent does, (24) will be incur fewer I/Os than (23) no matter what join algorithm is used to evaluate (23). If `Dept` fits in main memory, so will `Temp`. (24) will scan `Dept` once to create `Temp` and load it into the buffer pool. Since `Temp` is already in the buffer pool, the `Professor`|×|`Temp` loop will only require a single scan of the `Professor` extent, and the final scan of `Temp` will not incur any I/Os. Thus, query (24) requires only a single scan of the `Dept` and `Professor` extents. Query (23), like queries (16) and (18), must reread part of the `Professor` extent no matter what join algorithm is used. Thus, query (24) will incur fewer I/Os than (23).

6. System Architecture of the Implementation

We implemented these ideas in the *O++* compiler being developed at Bell Labs-Murray Hill. The *O++* compiler is a *cfront 2.1 C++* compiler extended to handle database programming language constructs. Like *cfront*, it parses a single program unit (e.g. a variable declaration, a `typedef`, or a function definition). The compiler then passes the parse tree to a print routine that emits *C++* code corresponding to the *O++* code that was just parsed (all calls to the underlying storage manager are encapsulated in *C++* classes). It then parses the next program unit, and so on. While processing a program unit, it also makes symbol table entries.

6.1. Optimizer

To add our optimizations to the *O++* compiler, we modified the print routine of the set loop construct. Instead of printing the parse tree itself, the print routine passes the parse tree to a routine that massages it into the query tree representation described in Section 4. This new tree is then passed to our optimizer.

The optimizer was built using the EXODUS Optimizer Generator, which was the only publicly available tool that allowed us to build an optimizer without having to write a search strategy. The transformations presented in the paper as well as standard relational transformations and several utility transformations (for instance, transformations to ensure that the tree rewrites required to produce *Tb'* for (T4) in Figure 7 are performed) were encoded in the Optimizer Generator's rule syntax. The Optimizer Generator transformed the rules into a set of routines that implemented the rules. While the rules could be expressed quite concisely, most rules required that we write a substantial amount of *C++* support code to handle the transfer of arguments from the original to the transformed tree. Using the Optimizer Generator, it was easier to add new transformations than it would have been in a hand-coded optimizer. Each new transformation added at most two dozen lines to the rule file given to the Optimizer Generator. Support code had a particular form imposed by the Optimizer Generator, which made the coding more uniform (and hence comprehensible) than it otherwise might have been.

The optimizer produced by the Optimizer Generator takes an initial query tree as input and uses transformations to produce equivalent plans. The optimizer explores the space of equivalent queries searching for the cheapest plan. This plan is then passed to a routine that emits *C++* code corresponding to the optimized query plan.

6.2. System

We used a different run-time system than the one that the original *O++* compiler generated code for.[5] That system

[5] [AGRA91] contains a description of the original *O++* run-time system. It also contains examples of translating *O++* code into *C++* code.

swizzled all objects into main memory until the end of transaction. This works well if all the data needed by the transaction fits in main memory. However, our optimizations are intended to improve the performance of queries where only some of the data fits in main memory; they have only minimal impact on performance if all the data fits in main memory. Thus, we needed a different storage manager.

We used Version 1.2 of the EXODUS Storage Manager to hold our test database. In order to minimize the number of changes that needed to be made to the *O++* compiler, we wrote *C++* classes with interfaces that were very similar to the interfaces of the classes used by the original *O++* storage manager for accessing and creating EXODUS Storage Manager data. We didn't build the complete set of classes or interfaces provided by the *O++* storage manager, just enough to create data and to run read-only queries.

Since the current *O++* implementation does not provide indexes or any join algorithms other than tuple-at-a-time nested loops, we do not provide them either. This implies that the major join optimization is to ensure that the innermost set fits in the buffer pool. If none of the sets of a group-by loop will fit in the buffer pool, even after they are selected and projected, our optimizer can do nothing to prevent the query from running quite slowly. We plan to extend the optimizer when indexes and new join methods are added. Since unanalyzed group-by loops must be evaluated with a tuple-at-a-time nested loops join algorithm (possibly with an index), and adding more facilities will improve the quality of optimized plans, these extensions should make the difference in performance between optimized and unoptimized plans even more impressive than in the examples in the following section.

7. Experiments

In this section, we consider two queries, and compare the performance of the optimized and unoptimized forms of each. The purpose of this section is not to exhaustively enumerate the types of queries that can be optimized, but rather to demonstrate that the ideas in this paper can be implemented and can significantly improve performance.

Our experiments were run on a DECstation 3100 with 20 megabytes of main memory and a 10 megabyte (2500 4K page) buffer pool. Each experiment was repeated three times and the observed response times were averaged. In the experiments, there were 100 `Dept`, 2000 `Professor`, and 3000 `Enroll` objects. The `Dept` extent contained 25 pages, the `Professor` extent 500, and the `Enroll` extent 3000. Each `Dept` had 20 `Professors`. Half the `Professors` taught one class; the other half taught two. The objects from each extent were clustered together. In this section, we show the original query and an *O++* representation of the *C++* code produced by the optimizer. The names created by the compiler for the optimized code were simplified by hand.

The first query optimized was:[6]

```
(25) for (P of Professor) suchthat (P->pid<upper && P->pid>=lower) {
         pcount++;
         for (E of Enroll) suchthat (P->pid==E->pid) {
             printf("%s %s %d ",P->Pname,E->name,E->studentcount);
             ecount++;
         }
     }
```

(25) is optimized by applying (T4) to pull the `pcount++` statement out of the `Professor` loop. (T4) also produces a selected, projected subset of `Professor` called $\texttt{Temp}_{\texttt{Prof}}$. Now there are two loops, the second of which is a simple group by loop over $\texttt{Temp}_{\texttt{Prof}}$ and `Enroll`. (T2) is then applied to the simple group by loop to produce:

```
(26) Temp_Prof = {};
     for (P of Professor) suchthat (P->pid<upper && P->pid>=lower) {
         pcount++;
         Insert <P->pid, P->Pname> into Temp_Prof;
     }

     Temp_join = {};
     for (E of Enroll)
         for (T of Temp_Prof) suchthat (T->pid==E->pid)
             Insert <T->Pname,T,E->name,E->studentcount> into Temp_join;

     Sort Temp_join by T->T_oid; //use oids from Temp_Prof to sort

     for (T2 of Temp_join) {
         printf("%s %s %d ",T2->Pname,T2->name,T2->studentcount);
         ecount++;
     }
```

Remember that tuple-at-a-time nested loops is currently the only join method in the system. In (25), the inner set `Enroll` is allocated 2482 pages out of a 2500 page buffer pool for pinning pages. These pages only need to be read once. The join must, however, read the remaining 518 pages once for each selected `Professor`. Let $\texttt{sel}_\texttt{P}$ be the number of selected objects from `Professor`. We would expect (25) to cost

500	pages of `Professor`
+2482	pages of `Enroll` that can be pinned in main memory
$+518 \times \texttt{sel}_\texttt{P}$	pages of `Enroll` that cannot be pinned

page reads. Query (26), on the other hand, reads `Professor` once. Since both $\texttt{Temp}_{\texttt{Prof}}$ and $\texttt{Temp}_{\texttt{join}}$ fit in the buffer pool, the join that produces $\texttt{Temp}_{\texttt{join}}$ requires only a single scan of the `Enroll` extent. Since $\texttt{Temp}_{\texttt{join}}$ fits in the buffer pool, neither the sort nor the final scan incur any I/Os. Thus (26) requires only a single scan of the `Professor` and `Employee` extents.

[6]A simple group-by loop similar to (25) was optimized as well. The resulting graph looked very similar to Figure 8, the graph for (25) and (26), so it was not included.

In Figure 8, the response time for queries (25) and (26) over a range of values of sel_P is presented. The selection predicate was chosen so that the number of elements in Professor|×|Enroll was 1.5 times the number of selected Professor objects. The timing information in Figure 8 comes from using the UNIX command gettimeofday before the transaction that evaluates the query begins and after it ends.

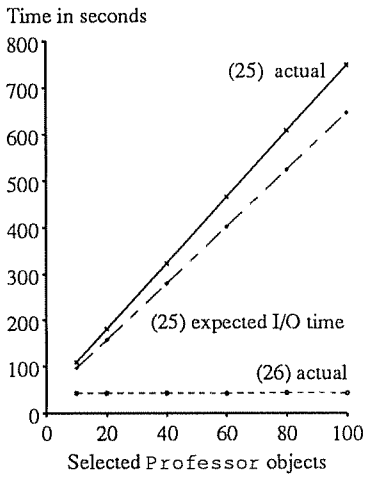

Figure 8

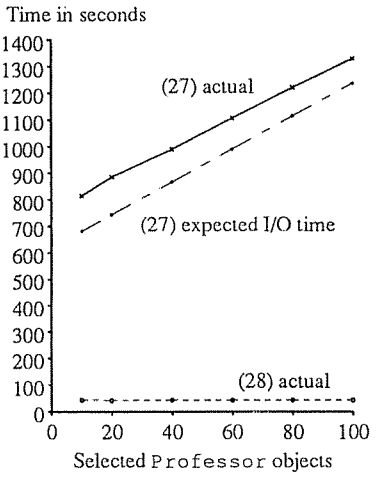

Figure 9

We include the expected I/O time required for query (25). We calculated the expected disk speed of the machine to be 84.674 pages/second by running a simple program that sequentially scans a large file by doing an lseek and then a 4K read. It repeats this sequence 4000 times. The getrusage and gettimeofday commands were used in the same way as above to produce this estimate. We did not include the estimate for (26), since it coincides with the actual time of (26) in the graph. It was actually about 1.5 seconds too low, but this difference didn't show up on the graph. The time for (26) is approximately flat across the range because the query is I/O bound, and the amount of I/O is the same for each value of sel_P since both Professor and Enroll are only read once, and $Temp_{Prof}$ and $Temp_{join}$ can be kept in the buffer pool. The actual run time for (25) is closely approximated by the expected I/O time of the query. The actual run time is somewhat higher for at least two reasons. First, the expected I/O time is based on sequential reads, and (25) requires some random I/Os. Second, (25) causes many more page reclaims (i.e. page faults that find the desired page in a list of pages the UNIX kernel plans to write to disk) than (26) does (the estimating program caused no page reclaims). Both these costs increase as sel_P increases, which is why the slope of the actual response time curve is steeper than the slope of the the expected I/O time curve.

The second query optimized was:

```
(27) for (D of Dept) {
         deptcnt++;
         for (P of Professor) suchthat
         (P->did==D->did && P->pid<upper && P->pid>=lower) {
             studentstaught = 0;
             for (E of Enroll) suchthat (E->pid==P->pid)
                 studentstaught += E->studentcount;
             printf("%s %s %d",D->Dname,P->Pname,studentstaught);
         }
     }
```

The optimized form of (27) used (T4) to move `deptcnt++` out of the main loop and to project `Dept` to produce a temporary set, $Temp_{Dept}$. It then applied (T5) to produce:

```
(28) Temp_Dept = {};
     for (D of Dept) {
         deptcnt++;
         Append <D->did,D->Dname> to Temp_Dept;
     }
     Temp_join = [];
     for (TD of Temp_Dept)
         for (P of Professor) suchthat
         (P->did==TD->did && P->pid<upper && P->pid>=lower) {
             studentstaught = 0;
             Append <TD->Dname,P->Pname,P->pid,studentstaught> to Temp_join;
         }
     for (E of Enroll)
         for (T of Temp_join) suchthat (E->pid==T->pid)
             T->studentstaught += E->studentcount;
     for (T of Temp_join)
         printf("%s %s %d",T->Dname,T->Pname,T->studentstaught);
```

The timing results are shown in Figure 9. In (27), the inner set `Enroll` is allocated 2477 pages for pinning pages in the buffer pool. The remaining 523 pages must be reread sel_P times. Since there are no remaining pages for pinning `Professor` pages, `Professor` must be read once for each `Dept`. Query (27) must also read `Dept` once. Query (28), on the other hand, reads the `Dept` extent once. Since $Temp_{Dept}$, $Temp_{join}$, and `Professor` can all fit in the buffer pool at the same time, the second loop in (28) requires only a single scan to pin all the pages of `Professor`. Since $Temp_{join}$ is already in the buffer pool, the third loop costs only a single scan of `Enroll`, and the final loop incurs no I/Os. Thus, (28) requires only a single scan of the `Dept`, `Professor`, and `Enroll` extents. The curve for (28) is almost flat because (28) is I/O bound and the amount of I/O is the same for all values of sel_P.

8. Conclusions and Future Work

This paper demonstrates that standard transformation-base optimizer technology can be used to optimize group-by loops in a database programming language. An optimizer built using the EXODUS Optimizer Generator was added to

the Bell Labs *O++* compiler. The resulting optimizing compiler was used to experimentally validate the ideas in this paper. The experiments show that this technique can significantly improve the performance of database programming languages.

Future work includes finding new transformations, particularly transformations that combine several loops that appear sequentially in the program text into a single large loop (in some ways finding an inverse of transformations (T3) and (T4)–closely related to multi-query optimization [SELL88]). Pointer-based join optimizations [SHEK90] will also be explored. We are very interested in techniques for optimizing more complicated set loops–particularly loops that employ an *O++* `by` clause or its equivalent. (The `by` clause allows a user to specify the iteration order for a set loop. For instance, the user might specify iterating through the `Dept` extent in alphabetical order of the `Dept` names.) Having seen parallels between this work and work on parallelizing FORTRAN, we would also like to determine whether or not our analysis will enable the parallelization of sequential set iteration code written in a database programming language.

Acknowledgements

We are thankful to Bell Labs-Murray Hill for providing us access to the *O++* compiler and for supplying a summer of funding for the first author. The bulk of the implementation effort discussed in this paper was done during this summer at Bell Labs. We are indebted to Narain Gehani and Shaul Dar for helping us understand the compiler. Without their help, the implementation would have taken much longer to complete.

9. Bibliography

[ABU81] Walid Abu-Sufah, David J. Kuck, and Duncan H. Lawrie. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. *IEEE Trans. on Computers C-30,5* (May 1981), 341-355.

[AGRA89] R. Agrawal and N. H. Gehani. Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language *O++*. *Proc. 2nd Int. Workshop on Database Programming Languages*, June 1989.

[AGRA91] R. Agrawal, S. Dar, and N. H. Gehani. The *O++* Database Programming Language: Implementation and Experience. AT&T Bell Labs Technical Memorandum, 1991.

[ATKI89] Malcolm P. Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The Object-Oriented Database System Manifesto, invited paper, *1st Int'l Conf. on Deductive and Object-Oriented Databases*, Japan, December, 1989.

[DAYA87] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Process Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. *Proceedings of 1987 Conf. Very Large Databases*, August 1987.

[DEMO85] G. Barbara Demo and Sukhamay Kundu. Analysis of the Context Dependency of CODASYL FIND-statements with Application to Database Program Conversion. *Proc. 1985 SIGMOD*, May 1985.

[DEWI84] David DeWitt, Randy Katz, Frank Olken, Leonard Shapiro, Michael Stonebraker, and David Wood. Implementation Techniques for Main Memory Database Systems. *Proc. 1984 SIGMOD*, June 1984.

[GANS87] Richard A. Ganski and Harry K. T. Wong. Optimization of Nested SQL Queries Revisited. *Proc. 1987 SIGMOD*, May 1987.

[GRAE87] Goetz Graefe. Ph.D. Thesis. Rule-Base Query Optimization in Extensible Database Systems. University of Wisconsin (1987).

[KATZ82] R. H. Katz and E. Wong. Decompiling CODASYL DML into Relational Queries. *ACM Trans. Database Syst. 7,1* (March 1982), 1-23.

[KIM82] Won Kim. On Optimizing an SQL-like Nested Query. *ACM Trans. Database Syst. 7,3* (September 1982), 443-469.

[LECL89] C. Lecluse and P. Richard. The O_2 Database Programming Language. *Proc. 1989 Conf. Very Large Databases*, August 1989.

[LIEU91] Daniel Lieuwen and David DeWitt, Optimizing Loops in Database Programming Languages. *Proceedings of 3rd Int'l Workshop on Database Programming Languages*, August 1991.

[LOHM88] Guy Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. *Proc. 1988 SIGMOD*, June 1988.

[MURA89] M. Muralikrishna. Optimization and Dataflow Algorithms for Nested Tree Queries. *Proceedings of 1989 Conf. Very Large Databases*, August 1989.

[PADU86] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *CACM 29,12* (December 1986), 1184-1201.

[RIES83] Daniel Ries, Arvola Chan, Umeshwar Dayal, Stephen Fox, Wen-Te Lin, and Laura Yedwab. Decompilation and Optimization for ADAPLEX: A Procedural Database Language. Computer Corporation of America, Technical Report CCA-82-04, Cambridge, Mass., September 1983.

[RICH89] Joel Richardson, Michael Carey, and Daniel Schuh. The Design of the E Programming Language. Technical Report #824, Computer Sciences Department, University of Wisconsin, February 1989.

[SCHM77] Joachim Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Trans. Database Syst. 2,3* (September 1977), 247-261.

[SELL88] Timos Sellis. Multi-Query Optimization. *ACM Trans. Database Syst. 13,1* (March 1988), 23-52.

[SHEK90] Eugene J. Shekita and Michael J. Carey. A Performance Evaluation of Pointer-Based Joins. *Proc. 1990 SIGMOD*, May 1990.

[SHOP80] Jonathan Shopiro. Ph.D. Thesis. A Very High Level Language And Optimized Implementation Design For Relational Databases. University of Rochester (1980).

[WOLF86] Michael Wolfe. Advanced Loop Interchanging. *Proc. 1986 Int. Conf. Parallel Processing*, August 1986.

[WOLF89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, Massachusetts, 1989.

Appendix: Optimization Example

To give a better feel for how the optimizer works, we will trace the steps the optimizer goes through to transform a query to a more efficient form. The example will only consider the steps taken along a single transformation path; the optimizer will also consider a number of plans that are not included in this section. Consider the following query:

```
(29) for (D of Dept) {
         deptcnt++;
         for (P of Professors) suchthat (P->did==D->did) {
             studentstaught = 0;
             for (E of Enroll) suchthat (E->pid==P->pid)
                    studentstaught += E->studentcount;
             printf("%s %s %d",D->Dname,P->Pname,studentstaught);
         }
     }
```

The optimizer first transforms (29) into the tree representation that is shown in Figure 10 (the *IsSeq* and *SelectPred* fields are not included in our example trees).

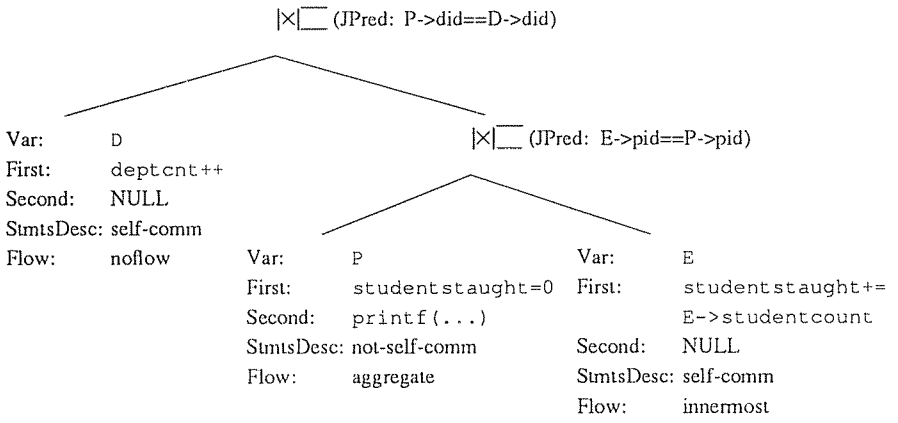

Figure 10: Tree Representation of (29)

The `printf` statement makes the `P` node *not-self-comm* since permuting `printf` statements will cause the program to produce a different output stream (i.e. the statement sequence `printf("a"); printf("b")` produces a different output stream than `printf("b"); printf("a")` does).

The optimizer notes that the `D` node's *StmtsDesc* field has the value *noflow*. It applies transformation (T3) to produce the tree in Figure 11:

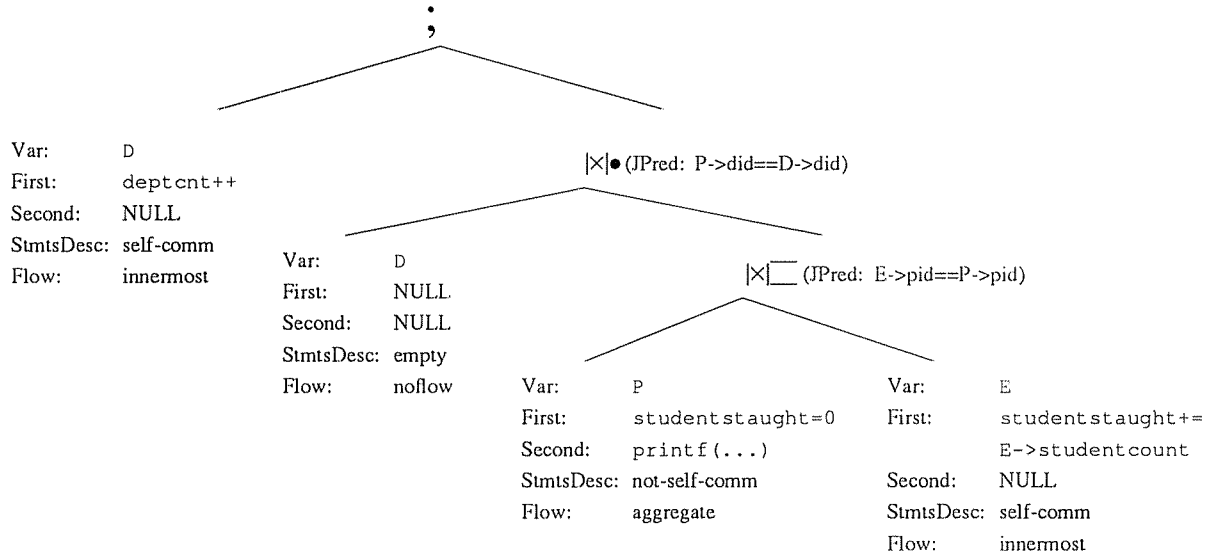

Figure 11: Tree Representation of (30)

which corresponds to the following code:

```
(30) for (D of Dept)
         deptcnt++;

     for (D of Dept)
         for (P of Professors) suchthat (P->did==D->did) {
             studentstaught = 0;
             for (E of Enroll) suchthat (E->pid==P->pid)
                 studentstaught += E->studentcount;
             printf("%s %s %d",D->Dname,P->Pname,studentstaught);
         }
```

The optimizer next considers the right-hand branch of the ; operator in Figure 11 (which corresponds to the second loop in (30)). Since the D node's *StmtsDesc* field has the value *empty*, it applies the Supernoding transformation to produce:

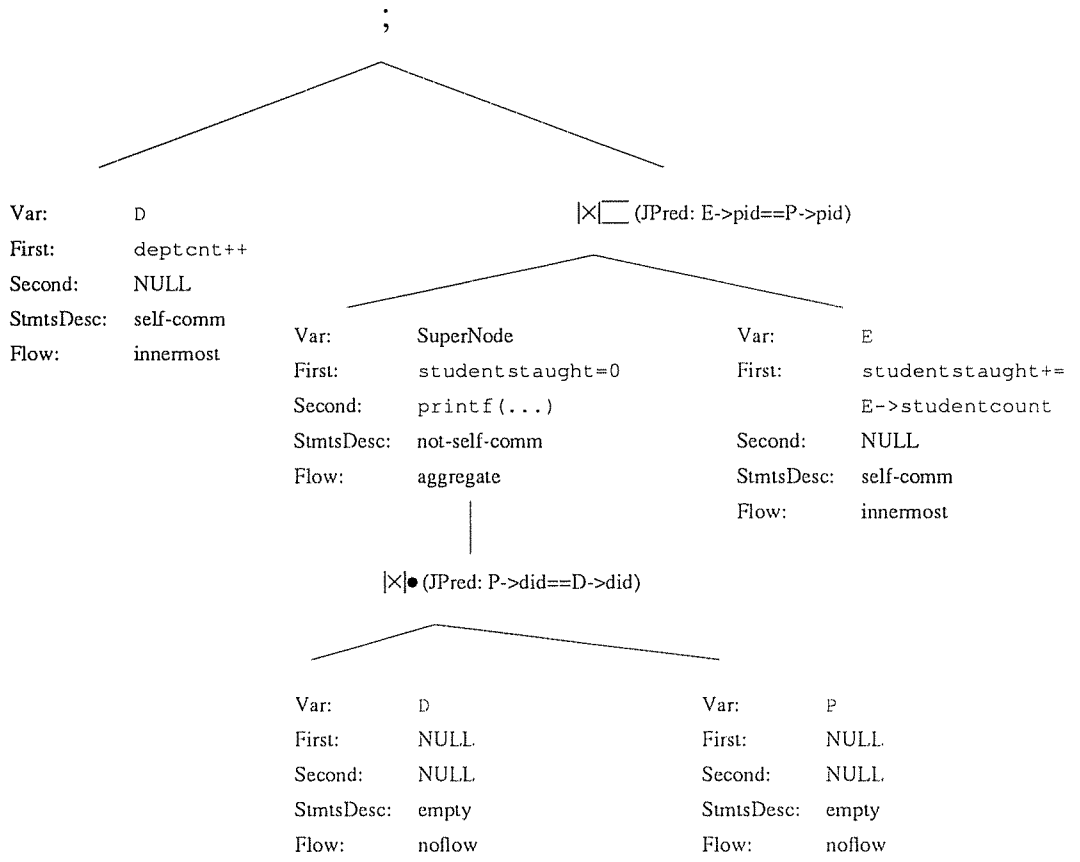

Figure 12: Alternative Tree representation of (30)

Figure 12 also corresponds to (30). The Supernoding transformation does not change the code that will be generated; it allows the optimizer to recognize that (T5) may legally be applied to the second loop in (30). After applying (T5) to the right branch of the ; operator in Figure 12, the following tree is produced:

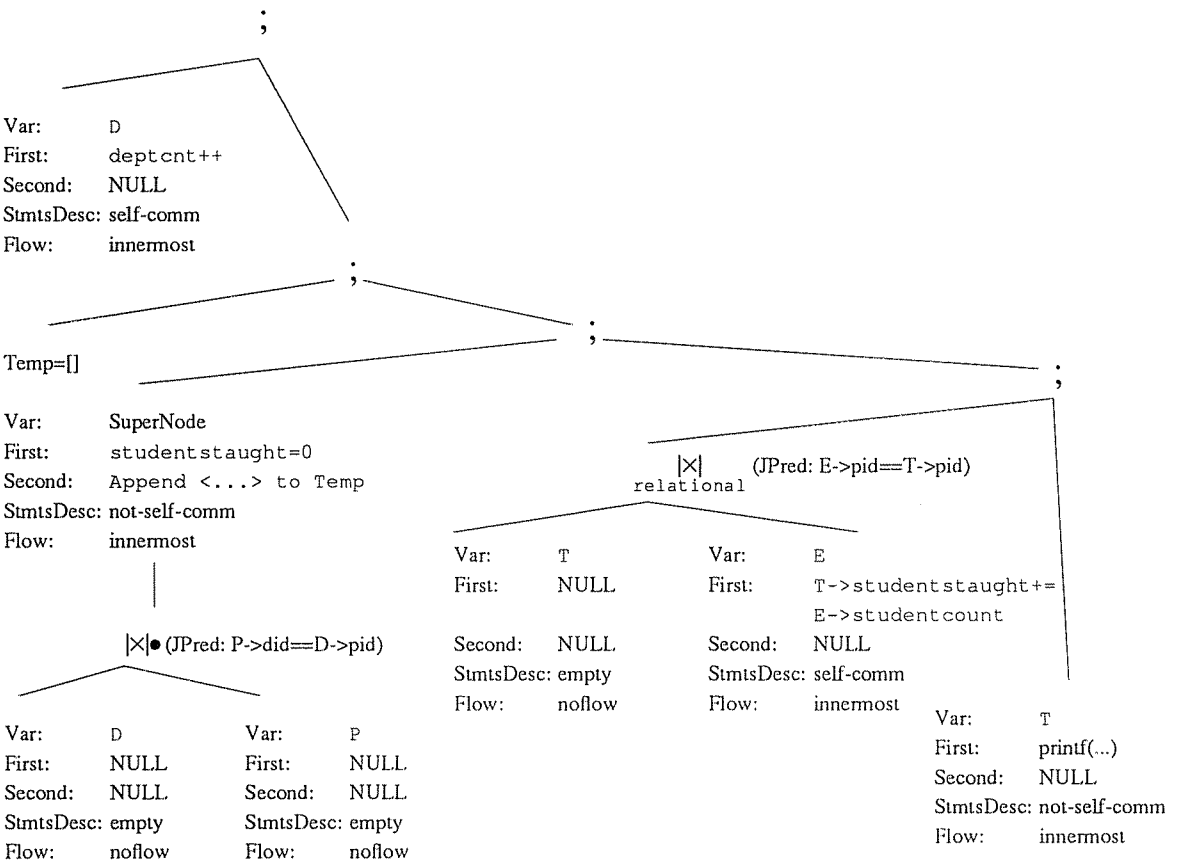

Figure 13: Tree representation of (31)

Figure 13 corresponds to the code fragment:

```
(31) for (D of Dept)
        deptcnt++;

     Temp = [];
     for (D of Dept)
        for (P of Professors) suchthat (P->did==D->did) {
           studentstaught = 0;
           Append <D->Dname,P->Pname,P->pid,studentstaught> to Temp;
        }

     for (T of Temp; E of Enroll) suchthat (E->pid==T->pid)
           T->studentstaught += E->studentcount;

     for (T of Temp)
        printf("%s %s %d",T->Dname,T->Pname,T->studentstaught);
```

The optimizer's final transformation of Figure 13 will be to choose to make `Temp` the inner set and `Enroll` the outer set for the $\underset{\text{relational}}{|\times|}$ subtree.