Optimizing Loops in Database Programming Languages

by

David J. DeWitt
Daniel F. Liewen

Computer Sciences Technical Report #1020
April 1991

Optimizing Loops in Database Programming Languages

Daniel F. Lieuwen
David J. DeWitt

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

April 2, 1991

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. We describe compile-time optimizations of such programming constructs that are similar to relational transformations like join reordering. Ensuring that the program's semantics are preserved during transformation requires paying careful attention to the flow of values through the program. This paper presents conditions under which such transformations can be applied and analyzes the I/O performance of several different classes of program fragments before and after applying transformations. The analysis shows that the transformations can significantly reduce the number of I/Os performed, even when both the initial and transformed programs use the same join method.

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 actually requires. Thus, compilers must be extended to include database-style optimizations.

It is especially important that compilers optimize join-like operators. These operators come from at least three sources. First, it is unlikely that there will be a pointer between every pair of objects that need to be related. 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. Second, the use of pointers leads to implicit joins. If we blindly follow pointers in the order specified by the user, the execution of the join may become unnecessarily slow [SHEK90]. Third, some join-like operations will result from calling functions from within a set iteration—since the functions may also iterate through sets.

We will concentrate on value-based joins. Database programming languages like PASCAL/R, O_2, Rigel, DBPL, *E*, and *O*++ [AGRA89,ATKI87,LECL89,RICH89] have constructs to iterate through a set in some unspecified order. The iterators may be nested to express joins. The following is an example of a nested iterator expressed in *O*++:

```
(1) for D in Dept {
        deptcnt++;
        for E in Emp suchthat (D->id == E->deptid) {
          D->print();
          E->print();
          newline();
        }
    } /* 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 `deptcnt++` and another set loop. Due to the enclosed statements, the method of producing `Dept|x|Emp` in (1) is more constrained than it would be in the relational setting—the join stream must be grouped by `Dept`. 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 only another set loop, we say the loop is a **simple group-by loop**. If the statement `deptcnt++` was removed from (1), query (1) would be a simple group-by loop.

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 for query optimization. Thus, it is useful to remove as many ordering constraints as possible so that the join computation can be reordered. 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. The transformations presented in Section 4 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.

This paper's transformations and analysis are designed for value-based joins (i.e., group-by loops). The transformations can also be used to improve the performance of queries that involve implicit joins through pointers, though the analysis presented here will not directly apply to this case. However, using a technique from [SHEK90], pointer dereferencing can be executed in a manner similar to value-based joins when the appropriate extents (sets of all objects of a particular type) exist. This is because, in an implicit join, the join predicate essentially compares the value of a pointer field in one set of objects with the object identifiers of the objects in another set. Given this view of pointer-based joins, the transformations and analysis presented here are indeed applicable. In the third case, the case where a nested function call also contains a set loop, the transformations and analysis can only be used if the function call is in-lined.

The remainder of this paper is organized as follows. First, in Section 2, we survey related work. Then, in Section 3, we consider program analysis that allows permuting inner/outer set relationships in simple group-by loops. We define a class of statements that we call **self-commutative**. If a simple group-by loop contains a self-commutative statement, it can be optimized like a relational join. In Section 4, we use 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. In Section 5, we carefully characterize the class of self-commutative statements. Our conclusions are contained in Section 6.

2. Related Work

The work most closely related to ours can be found in [SHOP80]; the transformation that [SHOP80] called **loop inversion** is explored in Section 4.3.3. Loop inversion is the only transformation that is carefully characterized in [SHOP80]—other transformations are illustrated with examples, but the conditions under which they are applicable are not stated.

Our work is similar to work done in [KATZ82,DEMO85] to decompile CODASYL DML into embedded relational queries. In [KATZ85], 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. [DEMO85] uses more sophisticated analysis to decompile a larger class of programs. Both our work and theirs tries to take an imperative program and make it as declarative as possible while maintaining the program semantics. 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 they ignored some semantic issues that are central in this paper. Since they only looked at DML statements and a few other COBOL commands that affect the flow of currency, they ignore grouping constraints. This is reasonable because COBOL DML has no way of expressing a join without grouping constraints—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 the work by Won Kim on transforming nested query blocks in SQL into equivalent queries with no nesting [KIM82]. The style of transformation is similar. He starts with a simple kind of nested query and shows how to transform it into a join query that does not have 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 untransformed query. These subqueries can be simplified further by other transformations. [DAYA87,GANS87,MURA89] corrected errors in Kim's technique by replacing joins with outerjoins. [DAYA87,MURA89] developed pipelining techniques that remove some of the temporary relations introduced by Kim's technique. We, too, break a complicated subquery into several parts. We combine the execution of a subquery with the partitioning phase of a hybrid hash join—which is similar to using pipelining.

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

```
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 improve paging performance dramatically [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*++ syntax for expressing a join. The SQL query

(2) **select**(D.all, E.all) **from** D **in** Dept, E **in** Emp **where** D.id=E.deptid

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

```
(3) for D in Dept, E in Emp suchthat (D->id == E->deptid) {
      D->print();
      E->print();
      newline();
    } /* 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 a class of statements that we will call **self-commutative**. Consider the following simple group-by loop:

```
(4)  for X1 in Set1 suchthat Pred1(X1)
       ...
         for Xm in Setm suchthat Predm(X1, ..., Xm)
           S;
```

Definition: The statement S in (4) is **self-commutative relative to** X_1, X_2, **...,** and X_m if any execution of

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

produces the same program state as executing (4).

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. $\exists i<j$ s.t. `Setj` $= X_i$->*set*). If such embeddings exist, the inner/outer relationships among sets must obey the partial ordering that if a set `Si` is embedded in an object of set `So`, then the join must have the set loop for `Si` inside the set loop for `So`.[1]

Few computations are likely to satisfy this definition in practice. Due to the finite precision of real arithmetic and the possibility of overflow or underflow, rearranging the loops may lead to a different result. Thus we provide two generalizations. A statement S is **self-commutative ignoring overflow** if it would be self-commutative on a machine where integer overflow (and underflow) errors cannot occur. A statement S is **self-commutative ignoring finite precision** if it would be self-commutative on an infinite precision machine. Note that if S is self-commutative ignoring overflow, it is also self-commutative ignoring finite precision, since overflow is the result of

[1] 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), (4) 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.

the finite representation of numbers. If S is self-commutative ignoring overflow and neither overflow nor underflow occurs at S in either the transformed or the non-transformed program, the same value will be computed by S in both. This is not true if S is only self-commutative ignoring finite precision. The values computed are likely to differ by a small amount.

We assume that the optimizer can be configured to **ignore overflow**, to **ignore finite precision**, or to **worry about everything**. A reasonable default value would be **ignore overflow**. From now on, we will use the term self-commutative to denote self-commutative ignoring overflow, self-commutative ignoring finite precision, or simply self-commutative depending on the optimizer's configuration.

To identify members of the set of self-commutative statements, we make the observation that any given execution of a loop of the form (4) where none of the predicates have side-effects will produce the same program state as the execution of some loop-free code segment.[2] Conceptually, we are unrolling the loops on a per execution basis. We then have a large number of S-like statements that vary only in the values for $X_1, \ldots X_m$. It may be that executing an arbitrary permutation of this sequence of S-like statements will produce the same program state as (4) for a given execution. If this is true for all possible program executions, then S is self-commutative.

Self-commutative statements can be identified with a simple test. If any two adjacent S-like statements can be permuted without changing the final computation of the program, S is self-commutative (since we can continue this process to produce an arbitrary permutation).

Examples include the computation of a sequence of aggregates. For example, in

```
(6) for D in Dept
      for E in Emp suchthat (D->id == E->deptid) {
        totpay += (E->basepay*D->profitsharing)/100 + ChristmasBonus;
        empcnt++;
    }
```

the statement sequence that increments totpay and empcnt is self-commutative ignoring overflow. Since integer addition is associative and commutative, ignoring the effects of overflow, an arbitrary pair of instantiations of

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

like

---

[2]If Emp = { [Joe,1], [Jim,2], [Ralph,1] } and Dept = { [Shoe,1], [Candy,2] } and there are persistent pointers E1-E3 to Emp elements and D1-D2 to Dept elements, then the execution of (1) that produces

Shoe 1 Joe 1
Shoe 1 Ralph 1
Candy 2 Jim 2

will produce the same program state as:

```
D1->print(); E1->print(); newline();
D1->print(); E3->print(); newline();
D2->print(); E2->print(); newline();
```

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

can be flipped without changing the final value of `empcnt` or `totpay` (assuming no overflow/underflow takes place). We call statements like those in (7) **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. A more complete description of the class of self-commutative statements will be presented in Section 5.

Two Other Uses of Self-commutativity

The notion of self-commutativity is valuable for compiler diagnostics. The compiler should flag statements in set loops that are (potentially) not self-commutative unless an order on the elements of the set has been specified. Otherwise, the result of the computation will (potentially) be non-deterministic.[3] Such statements are likely to be errors.

We also note that we cannot execute group-by loops using a blocked-nested loop join algorithm unless `S`, the inner statement, is self-commutative. To see this, suppose that this is not the case. Consider the execution of

```
(9) for D in Dept
      for E in Emp suchthat (D->id == E->deptid) {
         printf("%s %d %s",D->name,D->id,E->name); newline();
        }
```

under the assumption that two objects fit on a page and two buffer pages are available. Let `Emp` = { [Joe,1], [Jim,2], [Ralph,1] } and `Dept` = { [Shoe,1], [Candy,2] }. If (9) is executed using a blocked nested loops algorithm, the output sequence

Shoe 1 Joe
Candy 2 Jim
Shoe 1 Ralph

is produced. However, this violates the **`group by`** semantics of (9). Most other join algorithms including hybrid hash, index nested loops, and sort-merge will, however, produce legal groupings.

4. Optimizations

As illustrated by (9), nested iterators can be used to express a join with a grouping constraint. These grouping constraints can negatively impact performance by preventing the reordering of join computations, so it is useful to remove as many of them as possible. 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. The transformations presented in this section employ extra set scans, temporary sets, set sorting, and nested statement rewrites (for instance, rewriting the `printf` in (9) to use elements of a temporary set instead of elements of `Dept` and `Emp`) to enable more reorderings to be made without modifying the program's semantics.

---

[3]Examples of non-determinism resulting from non-self-commutatative statements can be found in Sections 4.3.1 and 5.1.

4.1. Simple group-by loops

The simplest loops that may be rewritten are of the form:

```
(10) for X1 in Set1 suchthat Pred1(X1)
       ...
         for Xm in Setm suchthat Predm(X1, ..., Xm)
           S;
```

If S is a self-commutative statement, then, by the definition of self-commutativity, (10) is equivalent to:

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

Even if S is not self-commutative, if S does not modify Set1-Setm or the variables used in the predicates, (10) can be rewritten as a join followed by a sort:

```
(T2) Temp = {};
     for X1 in Set1, ... , Xm in 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 Seti $\forall i\ 1 \le i \le m$ are replaced by uses of the fields of Temp. Needed(X_i) refers to the fields of Seti that are used in statement S or that are needed for sorting. It includes a unique identifier for each X_i object other than X_m 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. 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 Set1-Setm-1 in the same order each time, although it may behave as if it was iterating through Setm 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.

4.1.1. Analysis of simple group-by loop optimizations

To simplify the analysis, we will analyze only the case where two sets are iterated over. We assume that disk reads will only be performed while reading Set1-Setm (i.e. S does not chase any disk pointers or iterate through any sets). In the analysis, we will refer to the inner set as R and the outer set as S. We assume that each element in the smaller set that satisfies the selection criterion will join with exactly k elements of the larger set and that each join operator is an equijoin.

To simplify the following analysis, we only analyze the I/O costs in this paper. Since the amount of CPU time required by each algorithm is roughly proportional to the number of I/Os performed, we do not believe that incorporating CPU costs would significantly change the results obtained. The notation in Table 1 will be used in the analysis below.

Name	Description
P	size of a disk page in bytes
M	number of memory buffer pages
$\|S\|$	number of objects in set S
$size_S$	size in bytes of an object in set S
$\pi width_S$	size in bytes of projected subobject
sel_S	predicate selectivity
O_S	number of objects per page, $O_S = \left\lfloor \frac{P}{size_S} \right\rfloor$
P_S	number of pages, $P_S = \left\lceil \frac{\|S\|}{O_S} \right\rceil$
F	size of a main memory hash table for S is $F \cdot P_S$

Table 1

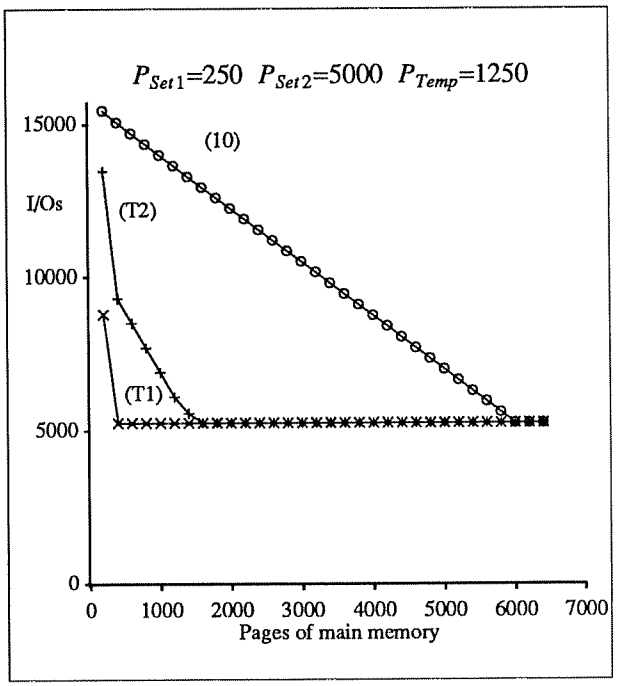

Figure 1

We will assume that queries are evaluated using the hybrid hash algorithm. [DEWI84] describes the hybrid hash algorithm as follows:

(1) Scan R, selecting those objects of R that satisfy the selection criterion, projecting out unnecessary attributes to produce R′. Objects of R′ are divided into $B+1$ partitions R_0, R_1, and R_B on the basis of a hash function applied to the join attribute. The same is done for S—this implies that R_i will only join with S_i $\forall i\ 0 \le i \le B$. The value of B is chosen so that: (1) S_0 can be joined in memory with R_0 as S is being partitioned, and (2) the hash table for R_j can fit in memory $\forall j\ 1 \le j \le B$. R_0 is kept in a main memory hash table; the R_j are written to disk.

(2) After S has been partitioned and S_0 has been joined with R_0, each R_i is joined with S_i $\forall i\ 1 \le i \le B$. R_i is read into memory, and each R_i object is hashed on its join value and inserted into a hash table. S_i is read into memory, and the join attribute of each S_i object is used to probe the hash table to find R_i objects that it joins with.

[DEWI84,SHAP86] provide the following analysis of hybrid hash's performance that we extend to include the effects of selection and projection. Let $B = \left\lceil \frac{(P_{R'} \cdot F) - M}{M-1} \right\rceil$ and $q = \frac{M-B}{F \cdot P_{R'}}$ (q represents the fraction of set R′

represented by R_0; $(M-B)$ is the amount of space available for the hash table for R_0, while $(F \cdot P_{R'})$ is the size of a hash table for all of R′. Thus, q must be no larger than one and no smaller than zero, and must be adjusted accordingly in some cases). The cost of executing a query using the hybrid hash algorithm is:

$P_R + P_S$	Read R and S
$(P_{R'}+P_{S'}) \cdot 2 \cdot (1-q)$	Writing and rereading hash partitions

Our formula for hybrid hash is similar to the one in [DEWI84]. There are only two differences: (1) following [SHAP86], we do not distinguish between random and sequential I/O, and (2) R′ and S′ replace R and S in the definitions of B and q and in the second line of the cost formula. For hybrid hash to be applicable, $M \geq \sqrt{P_{R'} \cdot F}$ and one page must always be available for reading the next page into—this page is not included in the M pages that we have direct control over. Note that the smaller R′ is, the larger q will be, and a larger q means that a smaller number of page I/Os will be performed. Thus, the selection of the inner and outer sets can have a significant impact on the execution time of this algorithm. In general, the smaller set should be the inner set.

Here, the concept of self-commutativity becomes useful because the optimizer cannot choose the probing set in (10) (as (T1) allows) unless the statement S is freely-permutable. Figure 1 compares the performance of the untransformed simple group-by loop (10) and the optimized query (T1) when $P_{Set1} = 250$ and $P_{Set2} = 5000$. Following [DEWI84], we assume $F = 1.2$. The size of memory was increased in 200 page increments until increasing the memory size did not change performance. The curves flatten when the inner set's hash table fits in main memory, since then each set is read only once. Using self-commutativity analysis to allow the smaller set, Set1, to be used as the inner set dramatically decreases the cost of the query.

Next, we analyze the effectiveness of transformation (T2). Note that it would be pointless to use transformation (T2) unless it permits the use of a more efficient join method or it allows Set2 to become the outer set. We consider only the second case. The cost of the first loop in (T2) can be determined using the analysis above if M is replaced by M_1=(M-1) (since one page is needed to hold Temp objects). By our hypothesis that each element of the smaller set, Set_{small}, that satisfies the selection criterion joins with exactly k elements of the larger set, |Temp| = $(|Set_{small}| \cdot sel_{Set_{small}} \cdot k)$. Note that $size_{Temp} = (\pi width_{Set1} + \pi width_{Set2})$. Given this, P_{Temp} can be calculated using the formula in Table 1. Assume that $M \geq \sqrt{\frac{P_{Temp}}{2}}$. Using the analysis found in [SHEK90], this implies that Temp can be sorted in two passes. Thus, the cost to execute the query resulting from transformation (T2) is:

join cost of `Set2`|X|`Set1` with $M_1 = (M\text{-}1)$	Join using `Set2` as the outer set
+UnbufferedPages(P_{Temp})	Write pages of `Temp` that cannot be buffered during join
+UnbufferedPages(P_{Temp})	Read pages of `Temp` written during join for first pass of sort
$+P_{Temp} \cdot 2$	Write runs, and read them back for sort
$-\min(P_{Temp}, M - \sqrt{\frac{P_{Temp}}{2}}) \cdot 2$	I/O savings if extra memory available during sort

The memory resident hash table for `Set1` has first claim on memory during the join, so if $(P_{Set1} \cdot F) \geq M_1$, all of `Temp` must be written to disk during the join. If, however, there is enough room for a hash table for all of `Set1`, any leftover pages can be used to buffer `Temp`. There will then be $(M_1 - P_{Set1} \cdot F)$ pages available for buffering, so only $(P_{Temp} - (M_1 - P_{Set1} \cdot F))$ pages must be written to disk. This formula is too simplistic if main memory can hold both `Temp` and a hash table for all of `Set1` with room to spare, because then this formula will calculate a negative number. It must, therefore, be changed to $max\,(0,(P_{Temp} + P_{Set1} \cdot F - M_1))$. Thus,

$$\text{(11) UnbufferedPages}(P_{Temp}) = \begin{cases} P_{Temp} & \text{if } (P_{Set1} \cdot F) \geq M_1 \\ max\,(0,(P_{Temp} + P_{Set1} \cdot F - M_1)) & \text{otherwise} \end{cases}$$

In Figure 1, we graph the performance of queries (10) and (T1) against the performance of the query produced by transformation (T2), assuming that $P_{Temp} = 1250$. The initial drop in the cost of executing (T2) occurs because the cost of the join with `Set1` as the inner set drops when `Set1`'s hash table fits in main memory. The smaller subsequent reductions are due to the buffering of more pages of `Temp` during the join and sort. The curve flattens when `Set1`'s hash table and the set `Temp` both fit in main memory. Note that the performance of the query after it has been rewritten using transformation (T2) is between that of (10) and (T1). This is not surprising since transformation (T2) uses hybrid hash with `Set2` as the outer set and then sorts `Temp`, the result of the join. Since `Temp` is only one fourth the length of `Set2`, this is less expensive than using `Set2` as the inner set. Thus, transformation (T2) can be an attractive option for executing (10) queries when statement sequence `S` is not self-commutative.

4.2. General Group-by Loops

The group-by loops exemplified by (10) 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. For example, the following query

```
(12) for X in Set1 suchthat Pred1(X) {
        S1;
        for Y in Set2 suchthat Pred2(X,Y)
          S2;
        S3;
     }
```

exemplifies the general case.

In the following sections, unless stated otherwise, we will assume that none of the statements `S1-S3` insert (delete) elements into (from) `Set1` or `Set2`. We further assume that `S1` and `S3` do not change the values of

elements of Set2 or the values of elements of Set1 other than the current X.

4.3. Optimizing General Group-by Loops

If a variable v is defined in the inner and the outer loop and both definitions reach a use outside the inner loop, we will say that the two loops **interfere**. If the loops interfere, (12) cannot be rewritten unless the conditions in Section 4.3.3 are met. The following is an example of interference:

```
(13) for D in Department  {
        cnt = 0;  //S1
        for E in Employee suchthat (D->id == E->deptid)
              cnt++;  //S2
        printf("%s %d", D->name, cnt); newline(); //S3
     }
```

Since the definitions of cnt in S1 and S2 both reach the use of cnt in the printf statement, the two loops interfere. Example (13) can be rewritten using the technique described in Section 4.3.3. However, even when interference prevents transformation, any join method that maintains the proper grouping semantics (any standard join method except blocked-nested loops) can be used to evaluate a group-by loop, but Set2 must be the inner set for the join.

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

If values do not flow from the inner to the outer loop or from the outer to the inner loop, and the loops do not interfere, (12) can be rewritten as:

```
(T3) for X in Set1 suchthat Pred1(X) {
        S1;
        S3;
     }
     for X in Set1 suchthat Pred1(X)
        for Y in Set2 suchthat Pred2(X,Y)
          S2;
```

Actually, the conditions must be tightened to require that at least one of the following three conditions hold: (1) Set1 is iterated over in the same order in both set loops; (2) the sequence S1;S3 is self-commutative relative to X; or (3) S2 is self-commutative relative to X and Y. Requiring that one of these conditions hold is necessary to avoid subtle forms of inconsistency. Consider:

```
(14) t1 = new Tree; t2 = new Tree;  //create empty trees
     for X in Set1 {
        t1->Insert(X->i);          //Not self-commutative
        for Y in Set2
             t2->Insert(X->i);     //Not self-commutative
     }
```

Suppose Set2 has exactly one element. Then, since there are no predicates, (14) will produce the same program state as:

```
(15) t1 = new Tree; t2 = new Tree;
     for X in Set1 {
        t1->Insert(X->i);
        t2->Insert(X->i);
     }
```

However, if (14) is transformed using (T3) to:

```
(16) t1 = new Tree; t2 = new Tree;
     for X in Set1  //iterate in order {[1],[2]}
        t1->Insert(X->i);

     for X in Set1  //iterate in order {[2],[1]}
        for Y in Set2
              t2->Insert(X->i);
```

an inconsistency may occur. Suppose that `Set2` has exactly one element, and that we iterate through `Set1` in the order {[1],[2]} in the first set loop and in the reverse order in the second set loop. Then after the execution of (16), `t1` will be a tree with the value one in the root node, and `t2` will have the value two in the root node. This is inconsistent with the program semantics. However, a smart storage manager might produce this iteration order if steps are not taken to prevent it. Since the order of iteration through a set is unspecified, the storage manager might produce set elements in different orders in both iterations, because it might have some pages of the set still resident in main memory. Rather than produce objects in the same order, it could first iterate through objects still in main memory and then go out to disk. This must be prevented if transformation (T3) is to be applied unless either condition (2) or (3) is met. This problem will only occur when there are non-self-commutative statements of similar form in the inner and outer loop bodies. Thus, if either condition (2) and (3) is met, this problem will not occur.

We expect that one of the three conditions will almost always be satisfied, so that transformation (T3) can be applied. The group-by loop produced by (T3) may be optimized using the techniques described in Section 4.1.

Analysis and Example of Flow Dependence Free Optimization

Transformation (T3) is beneficial if `S1` or `S3` chase disk pointers or iterate through a set. However, we will only analyze the case where transformation (T3) is used to make `Set2` the outer set. Since `Set1` then becomes the inner set, scanning `Set1` for the first set loop can be combined with the partitioning of `Set1` in preparation for the execution of the join loop. Thus, the I/O cost of the query resulting from transformation (T3) is:

`Set2` |×| `Set1` Join with `Set2` as the outer set. Combine scanning and partitioning `Set1`.

Note that if the outer loop had been a join loop instead of a set scan, this analysis is too simplistic. For instance, for

```
(17) for A in Seta, B in Setb suchthat Pred1(A,B) {
        S1;
        S3;
     }
     for Y in Set2 suchthat Pred2(X,Y)
         for A in Seta, B in Setb suchthat Pred1(A,B)
               S2;
```

it is likely that all the buffer pages will be needed for computing `Seta`|×|`Setb`. Thus, the scanning and partition-

ing will probably have to be done in separate passes. A formula that covered join loops would have to include the cost of producing the join Seta|×|Setb and then of writing the result of the join to disk so it could be used by the second set loop in (17). Similar comments can be made about future transformations where the analysis assumes that scanning and partitioning are performed simultaneously.

As an example of transformation (T3), consider:

```
(18) for Dept in Department {
        Companybudget += Dept->budget;
        for Emp in Employee suchthat Dept->id == Emp->deptid {
             cnt[Dept->floor]++; /* Calculate number of emps/floor */
             if( Dept->sales > (CompanySales/5) )
                 High_Sales_dept_sals += Emp->salary
        }
     }
```

Note that the statement sequence in the Employee loop is self-commutative. Using (T3), query (18) can be transformed to the following (skipping a few intermediate steps):

```
(19) for Dept in Department
        Companybudget += Dept->budget;

     for Emp in Employee
         for Dept in Department  suchthat Dept->id == Emp->deptid {
             cnt[Dept->floor]++; /* Calculate number of emps/floor */
             if( Dept->sales > (CompanySales/5) )
                 High_Sales_dept_sals += Emp->salary
         }
```

Hybrid hash can be used for both (18) and (19). This is clear for (19), but (18)'s implementation is somewhat more complex. Instead of using hybrid hash to compute the join for (18), it must be used to compute the outer join. An outer join is required to ensure that Companybudget will be incremented once for each Department, even for Departments with no Employees. The group-by loop in query (18) must be treated like:

```
//Compute the outerjoin and not the join Department|×|Employee
for Dept in Department
   for Emp in Employee suchthat Dept->id == Emp->deptid {
     if (Dept has changed since previous time through loop)
        Companybudget += Dept->budget;
     if(Dept->id==Emp->deptid) { //if current Dept,Emp pair part of join
        cnt[Dept->floor]++;
        if( Dept->sales > (CompanySales/5) )
           High_Sales_dept_sals += Emp->salary
     }
   }
```

Suppose that Department consists of 50 pages, that Employee consists of 1000 pages, and that there are 300 buffer pages. We assume that F=1.2 and ignore the effects of projections. Using the analysis in Section 4.1.1, the set loop in (18) will cost 2633 page reads. However, the join in (19) will only cost 1050 page reads, since a hash table for all of Department will fit in main memory. We simultaneously scan Department for the first set loop while partitioning it for the join loop, so the total cost of (19) is 1050 I/Os.

Even if scanning and partitioning cannot be combined, the extra scan may not cost anything—multi-query optimization may eliminate the cost. For example, assume that `CompanyTotalSales()` is an in-line function to calculate the total sales of all `Departments`. Then the following code fragment

```
CompanySales = CompanyTotalSales();
for Dept in Department  //from (19)
   Companybudget += Dept->budget;
```

can be expanded to

```
(20)  int t = 0;
      Department * D;
      for D in Department
         t += D->sales;
      CompanySales = t;
      for Dept in Department
         Companybudget += Dept->budget;
```

Simple multi-query analysis, allows (20) to be rewritten as:

```
(21)  int t = 0;
      for Dept in Department {
         t += Dept->sales;
         Companybudget += Dept->budget;
      }
      CompanySales = t;
```

Since `CompanySales` is used in the inner loop of (18), the evaluation of `CompanyTotalSales()` could not have been pushed inside the `Department` loop of (18). Thus, transformation (T3) creates opportunities for multi-query optimization. This style multi-query optimization can be useful as a final clean-up pass after all the other transformations have been performed, since it can merge loops that result from transformations and from in-lining functions. Transformations like (T3) make it easier to determine which loops may be merged because they produce simpler loops.

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

Transformation (T3) works well if no information flows in either direction between the inner and outer loop. In statements of the form (12), 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 `Set1`; 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 (12) can be rewritten as:

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

     for X in Set1 suchthat Pred1(X) {
        S1;
        Append <Needed(X), v1, ... ,vn> to SetTemp.
        S3;
     }
     for  T in SetTemp  /* in insertion order */
        for Y in Set2 suchthat Pred2(T,Y)
              S2';
```

`S2'` is `S2` rewritten to use fields of `SetTemp` instead of v_i $\forall i\ 1 \le i \le n$ and instead of fields of `Set1`. `Needed(X)` contains the fields of `X` used in `Pred2(X,Y)` and `S2`. Note that if `Pred1(X)` is very restrictive or objects in `SetTemp` are shorter than objects in `Set1`, (T4) may be preferable to (T3) because the cost of storing and rereading the necessary objects might 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 `Set1`. If `S2` is self-commutative, `SetTemp` may be a set.

Analysis and Example of Optimization in the Presence of Flow Dependences

As noted above, the application of transformation (T4) makes sense under several conditions. However, we will consider only the first case, the case where `S2` is self-commutative. In this case, the two loops can be flipped. Note that $size_{SetTemp} = (\pi width_{Set1} + \sum_{i=1}^{n} sizeof(v_i))$, and that |`SetTemp`| = (|`Set1`|$\cdot sel_{Set1}$). Once again, we can combine scanning and partitioning, so the overall cost of (T4) will be:

P_{Set1}	Scan of `Set1`
+join cost of `Set2`|×|`SetTemp`	Join with `Set2` as outer set
$-P_{SetTemp}$	Combine scanning `Set1` with partitioning `SetTemp`

As an example of a case where rewrite (T4) is profitable, consider a slightly modified version of the previous example in which the floor of a `Department` is not directly stored in the `Department` object. Instead, it is contained in an object that contains information about the part of the building belonging to the `Department`.

```
(22) for Dept in Department {
        floor = Dept->buildinginfo->floor; //S1
        for Emp in Employee suchthat Dept->id == Emp->deptid
             cnt[floor]++; //S2
     }
```

Note that our analysis does not directly characterize this case since it does not cover persistent pointer dereferencing. However, the extensions necessary for this example are trivial. We can use (T4) and a loop interchange to rewrite (22) as:

```
(23) SetTemp = {}; //S2 is self-commutative
     for Dept in Department {
        floor = Dept->buildinginfo->floor;
        Append <Dept->id, floor> to SetTemp;
     }
     for Emp in Employee
          for T in SetTemp suchthat T->id == Emp->deptid
             cnt[T->floor]++; // Calculate number of emps/floor
```

Another alternative would be to use copy propagation to move `Dept->buildinginfo->floor` into the loop and eliminate the assignment to `floor`, producing a group-by loop with a self-commutative statement inside. We can then swap the loops to produce:

```
(24) for Emp in Employee
          for Dept in Department suchthat Dept->id == Emp->deptid
              cnt[Dept->buildinginfo->floor]++;
```

Assume that there are 300 buffer pages, that `Department` consists of 50 pages and 500 objects, and that `Employee` consists of 1000 pages and 10,000 objects. Suppose that the objects that `buildinginfo` points to are contained in a large set. Then, every time `buildinginfo->floor` is dereferenced, one page read is performed.

Given these assumptions, the original query (22) will incur 2633 page reads for the join and another 500 page reads for the dereferencing operations. Query (23) costs 50 I/Os to read `Department` and 500 I/Os for the dereferencing operations. Since $P_{\texttt{SetTemp}}$ is so small, `SetTemp` can be buffered in main memory, so the cost of the join will only be the 1000 I/Os to read the `Employee` set. Thus, the total cost of query (23) is 1550 I/Os. Alternative (24) is the worst plan. The dereferencing operations alone require 10,000 I/Os (assuming each `Employee` joins with exactly one `Department`).

4.3.3. 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:

(25) **select**(D.name, count(*)) **from** D **in** Department, E **in** Employee **where** D.id=E.deptid

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

```
(26) for D in Department  {
        cnt = 0;  //S1
        for E in Employee suchthat (D->id == E->deptid)
             cnt++;  //S2
        printf("%s %d", D->name, cnt); newline(); //S3
     }
```

We consider a transformation from [SHOP80] to rewrite queries involving aggregate functions such as (26). In

```
(27) for X in Set1 suchthat Pred1(X) {
        S1;
        for Y in Set2 suchthat Pred2(X,Y)
          S2;
        S3;
     }
```

suppose that `S2` is self-commutative relative to `X` and `Y`, and 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.[4] The v_1, ..., v_n must be assigned to during each pass through `S1`. In `S2`, they may be employed only 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 `Set1`. Finally, `Set2` must not be nested inside an object of `Set1`. If these conditions are met, (27) can be rewritten as:

```
(T5) Temp = {};
     for X in Set1 suchthat Pred1(X) {
        S1;
        Insert <Needed(X), v1, ..., vn> into Temp.
     }
     for Y in Set2
         for T in Temp suchthat Pred2(T,Y)
            S2';

     for T in Temp
        S3';
```

`Needed(X)` are the fields of `Set1` mentioned in `S2`, `S3`, or `Pred2(X,Y)`. `S2'` and `S3'` are rewritten versions of `S2` and `S3` that replace uses and definitions of v_i with uses and definitions of `T->`v_i. They also replace uses of fields of `Set1` with uses of the fields of `Temp` (i.e. `X->a` is replaced with `T->a`).

Analysis and Example of Optimizing Aggregate Computation

In analyzing (T5), note that $size_{Temp} = (\pi width_{Set1} + \sum_{i=1}^{n} sizeof(v_i))$ and that scanning `Set1` and partitioning `Temp` can be carried out simultaneously. Given this, the analysis is as follows:

P_{Set1}	Scan of `Set1`
+join cost of `Set2`|X|`Temp`	Join with `Set2` as outer set
$-P_{Temp}$	Combine partitioning of `Temp` with scan of `Set1`
$+\begin{cases} 2 \cdot P_{Temp} & \text{if Temp's hash table does not fit in main memory} \\ 0 & \text{otherwise} \end{cases}$	Write dirty pages of `Temp` during join and then reread for scan unless all of `Temp` can be buffered in main memory

Using transformation (T5), query (26) can be rewritten as:

[4]Actually, **commutative assignments** to variables v_1, ..., v_n are allowed. The phrase **commutative assignment** will be defined in Section 5.1.

```
(28) Temp = {};
     for D in Department  {
        cnt = 0;  //S1
        Insert <D->id, D->name, cnt> into Temp;
     }
     for E in Employee
        for T in Temp suchthat (T->id == E->deptid)
           T->cnt++; //S2'

     for T in Temp {
           printf("%s %d", T->name, T->cnt); newline(); //S3'
     }
```

Using the same parameters as before, the unmodified query (26) performs 2633 I/Os. The rewritten query (28) requires 50 reads to scan `Department`. `Temp` is inserted in a memory resident hash table during this scan. Thus, the join does 1000 reads, since only `Emp` needs to be read. The final processing of `Temp` does not incur any I/Os since `Temp` is already in main memory. Thus, the total cost of query (28) is 1050 I/Os.

4.3.4. Optimizing More Deeply Nested Loops Using Previous Rules

The transformations of the previous sections can be used repeatedly to optimize more deeply nested loops than we have considered. We will give an example where (T3) and (T5) are both used. Consider:

```
(29) for D in Department {
        deptcnt++;
        for P in Professors suchthat P->deptid==D->id {
             studentstaught = 0;
             for E in Enroll suchthat E->profid==P->id
                    studentstaught += E->studentcount();
             printf("%s %s %d",D->Dname,P->Pname,studentstaught);
        }
     }
```

`Enroll` objects contain an embedded set of students enrolled in a particular course section. The method `studentcount()` counts the number of students in the set. Transformation (T3) applies because no values flow from the `Department` outer loop to the inner loops or from the inner loops to the outer loop. Thus, (29) can be rewritten as:

```
(30) for D in Department
        deptcnt++;

     for D in Department
        for P in Professors suchthat P->deptid==D->id {
             studentstaught = 0;
             for E in Enroll suchthat E->profid==P->id
                    studentstaught += E->studentcount();
             printf("%s %s %d",D->Dname,P->Pname,studentstaught);
        }
```

Transformation (T5) can then be applied to the second loop to produce:

```
(31) for D in Department
        deptcnt++;

     Temp = [];
     for D in Department
        for P in Professors suchthat P->deptid==D->id {
           studentstaught = 0;
           Append <D->Dname,P->Pname,P->id,studentstaught> to Temp;
        }

     for E in Enroll
        for T in Temp suchthat E->profid==T->id
           T->studentstaught += E->studentcount();

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

Actually, the transformation employed was a variant of (T5), since `Temp` is a sequence and not a set in (31); `Temp` must be a sequence instead of a set because the printing must be grouped by `Department`.

Suppose $P_{Department}$=2, $P_{Professors}$=40, P_{Enroll}=4000, M = 250, and F=1.2. The `Enroll` objects are quite large because they each contain an embedded set. The original query (29) incurs 42 I/Os to evaluate `Department|×|Professors`. The result of the join is an 80 page sequence `Result1`. The evaluation of `Result1|×|Enroll` incurs 11,848 I/Os.

The rewritten query (31) will incur 2 I/Os scanning `Department`, which it will buffer in main memory. The size of the result sequence `Temp` will be somewhat larger than in the original query since each `Temp` object contains a `studentstaught` field. Assume P_{Temp}=90. Then `Temp`, `Department`, and `Professors` will all fit in main memory. Thus, the `Department|×|Professors` query will incur 40 I/Os to read `Professors`. Since a hash table for `Temp` will fit in main memory, the `Enroll|×|Temp` query will only incur 4000 I/OS. The final scan of `Temp` does no I/Os since `Temp` is cached in main memory. Thus, the rewritten query requires 4042 I/Os.

4.3.5. An Analogue to Relational Join Associativity

As demonstrated in relational optimization, associativity is a vital property for query optimization. The following query

```
(32) for X in Set1 suchthat Pred1(X) {
          S1;
          for Y in Set2 suchthat Pred2(X,Y) {
              S2;
              for Z in Set3 suchthat Pred3(X,Y,Z)
                   S3;
              S4;
          }
          S5;
     }
```

essentially specifies joining `Set1` with `Set2` and then joining the result of that join in a pipelined fashion with `Set3`. We present an associative rule for transforming (32) provided two conditions are met: (1) statements `S1`-

S5 do not modify Set2 or Set3, and (2) Set2 and Set3 are not nested in objects of Set1. Provided these two conditions are met, statements may make arbitrary changes to Set1 and the loops may interfere. To use (T3), (T4), or (T5) to transform query (32), conditions on the flow of values must be met. Thus, the associativity rewrite will sometimes be applicable when the others are not. The associativity rewrite is the following:

```
(T6) Temp = []; /* Empty sequence */
     for Y in Set2 suchthat Pred2'(Y) {
        for Z in Set3 suchthat Pred3'(Y,Z)
             Append <OID(Y), Needed(Y), Needed(Z)> to Temp;

        if(nothing was added to Temp in for Z loop)
             Append <OID(Y), Needed(Y), NULL> to Temp;
             /* Doing an outer join */
    } // (π_{OID,Needed(Y),Needed(Z)} (Outerjoin(σ_{Pred2'}(Set2),Set3))) group by OID

    for X in Set1 suchthat Pred1(X) {
        S1;
        last = NULL;
        for YZ in Temp suchthat Pred2''(X,Y) { // in insertion order
           if (last != YZ->OID)
              S2''; //Do S2'' if there is a new Y value
           if (YZ does not have NULL for Needed(Z) && Pred3''(X,Y,Z))
              S3''; //Do S3'' if there is a real Z value && Pred3''
           if (last != YZ->OID)
              S4''; //Do S4'' if there is a new Y value
           last = YZ->OID;
        }
        S5;
    }
```

Needed(Y) are the fields of Set2 mentioned in S2-S4 or used in Pred2(X,Y) or Pred3(X,Y,Z). The definition is similar for Needed(Z). Pred2'(Y) and Pred3'(Y,Z) are derived from Pred2(X,Y) and Pred3(X,Y,Z) by making them less restrictive. All clauses of the predicate that involve either X or variables written by S1-S5 are replaced by new clauses that do not mention them and that do not reject any values that would have passed the original predicates. If the original predicate does not have any NOTs, this can be done by replacing all such clauses with TRUE. The '' notation means that the statement or predicate has been rewritten to use YZ->a instead of Y->a and YZ->b instead of Z->b.

We would like a simple transformation like (Set1 |×| Set2) |×| Set3 → Set1 |×| (Set2 |×| Set3) where the left-hand side corresponds to (32). However, since S2 and S4 must be executed for every qualifying object of Set2, care must be taken. Ensuring that S2 and S4 are executed the proper number of times requires that no qualifying Set2 objects are lost. As a result, an outerjoin must be used. If S2 and S4 are null statement sequences, the check for NULL can be removed, the outerjoin can be replaced with a join, and OID(Y) will not need to be stored in Temp objects.

A sequence is used to ensure that S2, S3, and S4 are executed in the same order as they probably would have been in the unmodified query (32). Since Temp is a sequence, objects of Temp will be partitioned in insertion order. By maintaining the scan order within the partition—which will be done in a straightforward implementation

of hybrid hash—the second join in (T6) will be done in insertion order.

Analysis and Example of Associativity Rewrite

In this rewrite, $size_{Temp}$ = (sizeof(OID) + $\pi width_{Set2}$+$\pi width_{Set3}$). Assuming once again that exactly $k \geq 1$ elements of the larger set join with each element of the smaller set, Set_{small}, that passes the join criterion, |Temp| = ($|Set_{small}| \cdot sel_{Set_{small}} \cdot k$). Temp must be written to disk as it is produced, and so one page is required as an output buffer. Thus, the overall cost is:

join cost of Set2|×|Set3	Join with Set2 as the outer set
with M_1=(M-1) buffer pages	Need one page for Temp's output buffer
+P_{Temp}	Write pages of Temp to disk as they are produced
+join cost of Set1|×|Temp	Join with Set1 as the outer set

As an example of transformation (T6), consider a query to list employees by department:

```
(33) for D in Dept {
        D->print(); //S1
        for E in Emp suchthat (D->id == E->deptid)
           for J in Job suchthat (J->id == E->job) {
              E->print(); printf(" %s ", J->title); newline(); //S3
           }
     }
```

which can be rewritten as:

```
(34) Temp = [];
     for E in Emp {
        for J in Job suchthat (J->id == E->job)
            Append <E's fields, J->title> to Temp;
        if(No Job joined with E->job)
            Append <E's fields, NULL> to Temp;
     }

     for D in Dept {
        D->print(); //S1
        for T in Temp suchthat (D->id == T->deptid)
            if(T->title != NULL) {
               T->print(); printf(" %s ", T->title); newline(); //S3'
            }
     }
```

Note that in (34), since S2 and S4 are empty, the outerjoin and the check for NULL could have been eliminated. We included them for illustrative purposes. Suppose Department has 10 pages, Employee has 100 pages, Job has 20 pages, F=1.2, $size_{Department}$=2 · $size_{Employee}$, and the buffer pool has 31 pages. For (33), Department|×|Employee will be evaluated using 30 pages since one page will be used as an output buffer for the result of the join. The join will cost 283 I/Os using the analysis of hybrid hash. We assume that each Employee joins with exactly one Department. Since each Employee and Department is printed in its entirety, the objects resulting from this join will be three times longer than Employee objects, so 300 pages must

be written to disk. We then join this 300 page result with `Job`. Since `Job` will fit in a main memory hash table, this join will only require reading the `Job` and the results of the `Department|×|Employee` join once— so only 320 I/Os will be needed. The total cost of (33) is 903 I/Os.

For (34), since only the title is needed from `Job`, we will assume that $size_{Temp}$ is about 20% longer than $size_{Emp}$. Since each person has one job, P_{Temp} will be 120 pages. `Job` will fit in a main memory hash table, so the join will incur only 120 I/Os. Writing `Temp` will require another 120 I/Os. Computing `Department|×|Temp` will cost 344 I/Os. Thus (34) will cost 584 I/Os.

5. Self-commutativity

The notion of self-commutativity was extensively in Section 4. However, the class of self-commutative statements was never carefully characterized. We will now examine the class.

5.1. Self-commutative arithmetic operations

Self-commutativity is defined for a statement `S` relative to a group-by loop of the form

```
(35) for X1 in Set1 suchthat Pred1(X1)
       ...
         for Xm in Setm suchthat Predm(X1, ..., Xm)
           S;
```

We begin the identification of **self-commutative** statements by first considering individual statements that perform numeric computation. Throughout this paper we have used the *C/C++* convention that

```
v = v op E;
```

can be abbreviated

```
v op= E;
```

Some +=, -=, and *=, operator assignments are self-commutative ignoring overflow. Suppose the statement

```
(36) v op= f(X1, ... ,Xm,v1, ... ,vn);
```

is contained in `S`. This statement is self-commutative ignoring overflow if v_1, ... ,v_n are variables that are not modified by `S`, `f` is a mathematical function (an expression that produces a value and has no side-effects), and `v` is an integer variable that is not used in any loop predicates in (35). To see this, consider an example when the nesting is only one level deep (i.e. m=2).

```
(37) for D in Dept
       for E in Emp suchthat (D->id == E->deptid)
         totpay += (E->basepay*D->profitsharing)/100 + ChristmasBonus;
```

Since integer addition is associative and commutative, ignoring the effects of overflow, an arbitrary pair of instantiations of

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

like

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

can be flipped without changing the final value of `totpay` (assuming no overflow/underflow takes place). The same principle holds for an arbitrary statement of the form (36) and for deeper nestings. It is important to understand that operations of this form are sufficiently powerful to compute an arbitrary aggregate once the requirement that `v` be an integer variable is relaxed. We call operations of the form (36) **reductions** because they reduce a subset of a set or Cartesian product to a single value in an order independent manner. This is a natural extension to the concept of array reduction, a concept used in the optimization of programs for supercomputers [WOLF89].

If the right-hand side of (36) evaluates to a positive integer and `v` is known to be positive, a /= operation of the above form is also self-commutative ignoring overflow [BATE90]. (The result of integer division where one of the operands is negative is implementation dependent in the *C* language.) Also, the + and - operators for integers can be treated as inverses ignoring overflow (The * and / operators are not inverses since $0 = (1/3)*3 \neq (1*3)/3 = 1$). Inverses are important because `S` may contain several statements of the form (36) that involve the same `v`. If the operators used in these statements are identical or inverses, these statements are equivalent to a single statement of the form (36). For instance, the statement sequence `v+=E1; v-=E2` is equivalent to the single statement `v+= (E1) - (E2)`, which is of form (36). However, the statement sequence `v+=E1; v*=E2` is not equivalent to any statement of the form (36).

The +=, -=, *=, and /= operators for reals of the form (36) (where `v` is now a real variable that is not used in any loop predicates) are self-commutative ignoring finite precision. The argument is the same as the one for integers. The + and - operators can be treated as inverses ignoring finite precision, as can the * and / operators.

Actually, the rule for being self-commutative can be loosened a bit. We define a **commutative assignment** to be a statement of the form `r = f(`X_1`, ... ,`X_m`,`v_1`, ... ,`v_n`)` where `f` is a mathematical function and the v_i are either variables not modified by the loop or **temporaries**. A **temporary** is a variable that must be defined by a commutative assignment before any use in `S`. Also, no assignment to a temporary may reach any use outside of `S`.[5] Since temporaries are always defined before they are used in `S` and only the current iteration of the loop uses the value, **commutative assignments** are always self-commutative.

We can loosen our characterization of a reduction. The v_i in (36) can either be variables constant for the duration of the set loop or temporaries. Since the original characterization of (36) includes only self-commutative statements, the looser characterization includes only self-commutative statements as well. The two characterizations can be seen to be logically equivalent by considering the definition of a temporary to be a macro definition. For example,

[5] If the assignment reached outside the innermost loop and the loops were interchanged, this would not violate the program semantics, but this would violate the definition of being **self-commutative**. A different permutation of statements would lead to a different value reaching the use of the temporary. Thus, a different program state would be produced.

```
b = X1->i * 2; //b is a temporary
if (X1->j < 5)
   b = X1->j + 3;
x += b + 4;
```

is equivalent to:

```
x += ( (X1->j < 5) ? (X1->j + 3) : (X1->i * 2) ) + 4;
```

which is of the form (36). The presence of conditional statements makes the equivalent macro expression unintuitive, but our restrictions on predicates in Section 5.3 will ensure that each use of a temporary is equivalent to the use of a complicated expression of the proper functional form (involving the `?` operator). Thus, the looser characterization makes the operator assignment less complicated by using the temporaries, but does not increase the class of computations that are self-commutative.

The class of self-commutative statements also includes statements of the form (36) that use `max=` or `min=` as the operator. We could continue adding cases, but these rules are sufficient for optimizing standard computations over sets such as variance, standard deviation, and SQL-like aggregates. They are simple and easily eliminate statements that are not self-commutative like

```
v += v + f(X1);
```

and

```
v += f(X1); r += v;
```

These statements are admittedly strange—the first is equivalent to `v = v + v + f(X1)`. They are eliminated because `v` appears to the right of the `+=` operation; and `v`—which is not a temporary— is modified during the execution of the loop. Such statements are not self-commutative because they make explicit use of intermediate values of `v`. This causes values of X_1 from previous iterations of the loop to be used again during the computation made during the current iteration of the loop—which makes iteration order significant. For instance, in

```
(38)   v = 0;
       for X in Set1   //Set1 = {[1],[2]}
           v += v + X->i;
```

if we iterate in the order {[2],[1]}, `v` will have the value five at the end of the loop. If we iterate in the reverse order, `v` will have the value four. The first value in the iteration stream is used twice. Thus, if a statement make explicit use of `v` in `S`, that statement is not self-commutative. As can be seen from the examples, non-self-commutative numeric computations are likely to be rare in practice.

5.2. Other Self-commutative operations

Other sources of self-commutative operations are set insertion (into a set other than `Set1`, ..., or `Setm`) and deletion of a set element (from a set other than `Set1`, ..., or `Setm`). For the insertion or deletion statement to be self-commutative, the elements inserted or deleted must be of the form `f(`$X_1, \ldots, X_m, v_1, \ldots, v_n$`)` where `f` is a mathematical function and the v_i are either temporaries or variables not modified in `S`. Note that any of the v_i can be a set that is not modified by statements in `S`. Arbitrary read-only queries can be evaluated against the set v_i to

compute f.

Another source of self-commutative statements are some abstract data type (ADT) method invocations. For an ADT method invocation to be self-commutative, all parameters to the ADT method being invoked must be of the form $f(X_1, \ldots, X_m, v_1, \ldots, v_n)$ where f is a mathematical function and the v_i are either temporaries or variables not modified in S. Further, the optimizer must be aware that the method commutes with itself.

The ADT implementor must supply information about which operations logically commute, since there is no way a compiler can identify all and only such operations. The permutability of an operation is dependent on what the data structure is intended to represent. For instance, tree insertion is a self-commutative operation if the tree is used to implement a dictionary, since a dictionary does not have an operation that allows the program to find the position of an element in the tree. However, tree insertion is not self-commutative if the data structure is being used as a general tree, since tree traversal can produce this information. Thus, maintaining insertion order is unimportant in the first case, but crucial in the second.

5.3. Self-commutative sequences of operations

Suppose each individual statement in S is self-commutative when viewed in isolation. Then S is self-commutative if it satisfies some additional properties. Suppose variable v is defined by a statement of the form

(39) v op= f(X_1, ... , X_m, v_1, ... , v_n);

in S. Then all of the definitions of v in S must be reductions and the reduction operators must be either identical or inverses (i.e. v += E1 and v -= E2 are allowed, since this pair is equivalent to v += (E1)-(E2). However, v += E1 and v *= E2 are not allowed, because this pair cannot be expressed in form (39)). If this condition holds for all variables defined by reduction operations in S, we say that S is **self-commutative relative to reduction operations.**

If T is an ADT instance that invokes methods M_1, ..., M_p in S, then $\forall i\ 1 \le i \le p\ \forall j\ 1 \le j \le p$ M_i must commute with M_j. If the two adjacent S-like statements in an unrolled version of the program are swapped, then the M_i invocation in the second S-like statement will move before the M_1, ... , M_p invocations of the first. Thus M_i must commute with all of them. For example, a dictionary might have operations Insert and Reorganize, where the second operation does not affect the logical structure of the dictionary, but only improves the efficiency of dictionary lookup. In order for the statement sequence

(40) D->Insert(X_1->name); D->Reorganize;

to be self-commutative, an arbitrary pair of instantiations such as

```
D->Insert("Joe"); D->Reorganize();
D->Insert("Jim"); D->Reorganize();
```

must be interchangeable Thus Insert must commute with both Insert and Reorganize for the statement sequence (40) to be self-commutative. If this condition holds, we say that S is **self-commutative relative to ADT operations.**

Statements in S may insert elements into or delete elements from a set but not both unless we are absolutely sure that the insertion and deletion sets are disjoint. Otherwise, an unrolled version might insert an element in one S-like statement and delete it in the S-like statement that followed; in which case, the set will not contain the element upon completion. If the S-like statements were flipped, the final set would contain the element. If this condition holds, we say that S is **self-commutative relative to set operations.**

In summary, suppose that S is straightline code and that each statement in S is self-commutative when viewed in isolation. Then S is self-commutative if it is **self-commutative relative to reduction operations, ADT operations,** and **set operations.**

If S is of the form

```
(41) S1;
     if( b(X1,X2,...Xm,v1, ... ,vn))
        S2;
     S3;
```

then S is self-commutative if the sequence of statements S1; S2; S3 is self-commutative, b is a mathematical function, and the v_i are either temporaries or variables not modified by S. The rule for switch statements is analogous. The intuition is that a sequence of self-commutative statements can only lose this property by having statements added. Deleting statements leaves the sequence self-commutative. A conditional statement merely removes some statements from the final unrolled version.

6. Conclusions and Future Work

Database programming languages (DBPLs) like O_2, E, and O++ include the ability to iterate through a set. Nested iterators can be used to express joins with grouping constraints. These grouping constraints can negatively impact performance by preventing the reordering of join computations, so it is useful to remove as many of them as possible. In this paper, we defined a class of statements, the class of **self-commutative** statements, that do not inhibit loop interchange in an application program written in a DBPL. We used the concept of self-commutativity and some analysis of the flow of values through a program in a series of compile-time optimizations that are similar to familiar relational transformations like join reordering. The transformations presented employ extra set scans, temporary sets, set sorting, and nested statement rewrites to transform program fragments without modifying the program's semantics. We also analyzed the I/O performance of several classes of program fragments before and after these transformations have been applied. The results obtained demonstrate that the transformations can significantly reduce the number of I/Os performed, even when both the initial and transformed programs use the same join method.

There are several issues to pursue. While we have developed a number of transformations, we need to develop heuristics to decide which transformations may be useful for a given loop. In addition, we need to develop methods to 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]). We examined how this could be done as a clean-up pass, but we did not integrate it with the transformation process. Having seen

parallels between this work and work on parallelizing FORTRAN, we would like to determine whether or not our analysis will enable the parallelization of sequential set iteration code written in a DBPL. We would also like to determine whether a standard relational transformation-based optimizer can be extended to incorporate our ideas. We are currently examining ways to express the transformations presented in this paper in an algebraic manner. The strategy we are considering is to perform the analysis necessary to transform the program text into a tree of operators. Then, a transformation-based optimizer can apply the transformations to the tree instead of to the original program. The goal is to transform nested statements (like `S` which is transformed to `S'` by (T2)) only when absolutely necessary—to transform nested statements only for the query plan chosen by the optimizer. This should significantly speed up the optimization process by eliminating unnecessary work.

7. 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.

[ATKI87] Malcolm P. Atkinson and O. Peter Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys 19,2* (June 1987), 105-190.

[ATKI89] Malcolm P. Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The Object-Oriented Database System Manifesto, invited paper, 1st International Conference on DOOD (Deductive and Object-Oriented Databases), Japan, December, 1989.

[BATE90] Samuel Bates, private communication.

[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.

[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. *Proceedings of 1989 Conf. Very Large Databases*, August 1989.

[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.

[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.

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

[SHAP86] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Trans. Database Syst. 11,3* (September 1986), 239-264.

[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.