Optimizing and Parallelizing Loops in Object-Oriented Database Programming Languages

Daniel F. Lieuwen

Technical Report #1097

July 1992

OPTIMIZING AND PARALLELIZING LOOPS IN OBJECT-ORIENTED DATABASE PROGRAMMING LANGUAGES

by

DANIEL F. LIEUWEN

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN — MADISON
1992

This research was partially supported by a grant from AT&T Bell Laboratories, by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, and by a donation from Texas Instruments.

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. Without program analysis, such joins must be evaluated using a tuple-at-a-time nested-loops join algorithm, because otherwise program semantics may be violated. Ensuring that the program's semantics are preserved during transformation requires paying careful attention to the flow of values through the program. This thesis presents conditions under which such transformations can be applied.

This thesis then 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 AT&T Bell Laboratories' $O++$ [AGRA89] compiler. We used the resulting optimizing compiler to experimentally validate the ideas in this thesis in a centralized database setting. The experiments show that this technique can significantly improve the performance of database programming languages.

The techniques and analysis are then applied to parallelizing loops. The transformations can be used to produce better parallel code than using a straightforward parallelization of the nested iterators. Parallel algorithms for accessing nested sets are analyzed, and a new algorithm is presented.

ACKNOWLEDGMENTS

I would like to thank my parents for their love and the values they instilled in me, and Rev. Cliff Bajema for helping me to see all they have given me.

It has been my privilege to work with Professor David DeWitt. He significantly improved my technical writing and my presentation skills. He showed me how to convert the germ of an idea into a coherent paper. His accurate assessment of how far along I was enabled me to finish up at least one semester sooner than I thought possible.

I would also like to thank Professor Mike Carey for the initial idea that got me working on this topic and his help along the way. The other members of my committee—Marvin Solomon, Raghu Ramakrishnan, and Guri Sohi—gave me a number of ideas to pursue in the future.

I would also like to thank my siblings—Ruth, Melanie, Tim, Dave, and Julie—for their love and encouragement and my friends Wendy Stanford (as an undergraduate) and the Tookeys (as a graduate student) who helped drag me out of my shell and keep me sane.

There are many other friends I should mention from college and graduate school, but as I recently discovered while making a graduation party list, one always misses several who belong on a list of important people in one's life, so I will refrain. You know who you are. Thank you for your support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. MOTIVATION

Relational database systems are widely used to manage long-lived data. Since their query languages are non-procedural, the system has a great deal of latitude in how it chooses to compute a query result. To take advantage of this latitude, relational database systems use very complex optimizers that search a space of alternative query plans for the least expensive alternative [SELI79, JARK84]. The resulting plan will often be orders of magnitude faster than a naive implementation.

However, relational systems have two major drawbacks. First, relational query languages (e.g. SQL) are not computationally complete. Thus, some database computations must be performed using a conventional programming language. As a result, clumsy interfaces between two languages based on different paradigms must be used [ATKI87]. This is called the *impedance mismatch* [VOSS91]. Second, SQL's type system is inadequate for many applications. A user with a complex graph structure (for example, a VLSI chip design) that needs to be stored in the database must first encode the nodes and edges of the graph using flat records. This can be very burdensome. Converting the graph structure between the database and programming language format creates new opportunities for programming errors.

Thus, more powerful database systems are needed. Recently, a number of object-oriented database systems (OODBSs) with the full data modelling power of a conventional programming language have been developed [AGRA89, LAMB91, LECL89, RICH92]. However, unless the query language of the OODBS is computationally complete, some computations will still need to be performed using a conventional programming language. Since the language of the OODBS is more complex than SQL, this would require a very complex interface between the OODBS and the programming language. Thus, many researchers believe that an OODBS must include computationally complete language for data retrieval and modification [ATKI89]. Since an OODBS includes sets, computational completeness includes the ability to iterate through a set. However, giving programmers this power allows them to write programs that can be orders of magnitude slower than the desired computation should be. The programs can also be very difficult to parallelize. This is a major problem, since an OODBS will not be used unless its

performance is comparable to that of a relational system.

One of the most important factors in the performance of an OODBS will be how it handles joins, the computation of the relationship between multiple pieces of data in the database (e.g. finding all the professors in the history department to print a report, or finding all the subparts of an engine to calculate the manufacturing cost). Much more useful information can be obtained by examining relationships than by examining individual pieces of data. Thus, joins are a very important operation. Unfortunately, they can also be very expensive to compute. As a result, a large amount of research on relational databases has been aimed at computing joins efficiently [SELI79, JARK84, DEWI84, GERB86, SCHN89, MUMI90, DEWI92b]. For an OODBS to satisfy customer performance requirements, it must effectively optimize joins. This is more complex for the database programming language (DBPL) of an OODBS than for a relational system, because there are a number of different ways to express a join in such systems. In this thesis, we ignore the optimization of joins written in a non-procedural fashion. They can be optimized using standard relational techniques [SELI79] or the more complex techniques developed for OODBSs [SHAW89, BEER90, VAND91]. Instead, we will concentrate on set iterators.

Since DBPLs such as PASCAL/R [SCHM77], O_2 [LECL89], E [RICH92], and O++ [AGRA89] provide constructs to iterate through a set in some unspecified order, it is possible to nest iterators to express value-based joins. The following is an example of a nested iterator expressed in O++:

```
(1.1) for (D of Division) {
        divisioncnt++;
        for (E of Employee) suchthat (E->division==D) {
            D->print();
            E->print();
        }
      }
```

Unless analysis is used, query (1.1) must be evaluated using a tuple-at-a-time nested-loops join algorithm [SELI79]. One element of `Division` must be read. Then `divisioncnt` is incremented, and all the elements of `Employee` are scanned to see if any of them match with the `Division` element. For each match, some printing is performed. After the first `Division` element is processed, the same steps are repeated with the second, the third, and so on. This is almost certainly not the best way to perform this computation. This thesis will consider program analysis and transformations that allow the OODBS to rewrite a procedural specification like query (1.1) as an equivalent query with fewer constraints on how it executes.

1.2. OUTLINE

The remainder of this thesis is organized as follows. First, Chapter 2 surveys related work. Then, in Chapter 3, we consider program transformations for a DBPL, and demonstrate using analysis and examples that the transformations can improve program performance substantially in a centralized system. We also define and make extensive use of a class of statements that we call **self-commutative**. The concept of self-commutativity is used to identify nested iterators that can be optimized like relational joins. In Chapter 4, we characterize a subclass of self-commutative statements, and prove the correctness of the characterization.

In Chapter 5, the representation of the transformations of Chapter 3 as tree rewrites is described. This allows standard database-style transformation-based optimization to be performed. An optimizer based on this technique was built using the EXODUS Optimizer Generator and added to the AT&T Bell Labs *O++* compiler. The resulting optimizing compiler was used in experiments that demonstrate that these transformations can produce significant performance improvements in a centralized OODBS.

Chapter 6 considered how these transformations can be used to enhance parallelism opportunities. Examples of how the parallel **Hybrid-hash** join algorithm can be modified to execute program statements rather than produce result tuples were given. The conditions under which this modified **Hybrid-hash** algorithm can be used to execute a nested set iterator while maintaining program semantics were discussed. If these conditions are met, a join loop can be parallelized to the same extent as a relational join. We then described, analyzed, and evaluated several parallel pointer-based join algorithms, one of them new, all of them previously unanalyzed. Our conclusions and future work are described in Chapter 7.

CHAPTER 2

RELATED WORK

2.1. RELATED TRANSFORMATION WORK

The work most closely related to ours can be found in [SHOP80]; the transformation that [SHOP80] called **loop inversion** is explored in Section 3.2.3.3. Loop inversion is used to calculate aggregate functions more efficiently. It 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. We implemented a slightly more general version of **loop inversion** than the version described in [SHOP80].

[RIES83] uses an algebraic framework to optimize set loops in ADAPLEX. The algebra handles looping constructs more complicated than those covered in this thesis. However, this algebra does not allow breaking a nested set iterator loop into several loops, a key technique in this thesis. This algebra also does not recognize that certain nested loops are semantically equivalent to joins, an observation that our optimizer exploits.

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 a 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 was 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 nested set iterator 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 thesis. Since they only examined DML statements and a few other COBOL commands that affect the values of the currency indicators in the user work area, 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 program-

mer check the transformed program to see if it has the proper semantics. Our use of the concept of self-commutativity allows us to convert nested set iterators 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 thesis is also related to the work on transforming nested query blocks in SQL into equivalent queries with no nesting [KIM82]. The style of transformation is similar. [KIM82] 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. [MUMI90] uses this prior work to convert queries with nested query blocks into a series of queries and view definitions. It then applies the magic sets transformation to reduce the amount of work required to compute the desired result by eliminating useless work performed in some cases by the earlier techniques. We all share the goal of making queries more set-oriented to improve performance. We all use the strategy of breaking a complicated (sub)query into several parts and then further transforming the resulting subqueries. In this thesis, 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.

2.2. RELATED PARALLELIZATION WORK

Our parallelization work uses both transformations and pointer-based join techniques. We discuss the related work for each in turn.

2.2.1. Related Transformation Work

An enormous amount of work has been done on parallelizing relational queries (e.g. [GERB86, SCHN89, GRAE90, KITS90, WOLF90, HUA91, WALT91, DEWI92a]). Other work has been on parallelizing loops in FORTRAN (e.g. [PADU86, WOLF86, WOLF89]) and in LISP [LARU89]. All this work makes extensive use of program transformations.

[HART88, HART89] discuss their parallelizing compiler for FAD, a functional DBPL. They use analysis to determine if a program can correctly be executed in parallel—bringing all the data to a central site and executing the program there is the default. They target their techniques toward a parallel, shared-nothing architecture. Their optimizer explores the space of equivalent queries searching for the cheapest plan. Most optimizers only applies a transformation if analysis indicates that the transformation will produce an equivalent program. However, their optimizer will sometimes apply a transformation and then check (using abstract interpretation) that the two programs are equivalent.

2.2.2. Related Pointer-based join Work

In [VALD87], auxiliary data structures called *join indices* that can be used to speedup join processing are described. A join index for relations `R` and `S` essentially precomputes the join between those two relations by stor-

ing pairs of tuple identifiers (tids). Each pair contains a tid from both R and S such that the corresponding tuples join with one another. In a uni-processor system, the basic algorithm scans the index, reading the referenced tuples. [VALD87] compared the performance of join indices to the **Hybrid-hash** join algorithm, and showed that a more elaborate join algorithm using join indices could frequently produce better performance than **Hybrid-hash** in a uni-processor system. [OMIE89] compared the two algorithms in a parallel environment and showed that **Hybrid-hash** will almost always outperform join indices except when the join selectivity is very high.

[CARE90] describes an *incremental join facility* added to Starburst to enable a relational DBMS to efficiently handle many-to-one relationships. The set representation employed is similar to the representation provided by network database systems. The paper describes the results of an empirical performance study comparing the performance of a number of pointer-based join algorithms.

[SHEK90] assumes a different set representation—one where objects contain lists of other objects' oids. The paper describes, analyzes, and compares uni-processor versions of the pointer-based **Hybrid-hash** and **Hash-loops** join algorithms. [MEHT91] describes parallel versions of both these algorithms. [MEHT91] also analyzes three parallel versions of **Hybrid-hash** and compares their performance.

Other proposed pointer-based join algorithms include pointer-based nested loops [SHEK90], pointer-based sort-merge [SHEK90], and pointer-based PID-partitioning [SHEK91]. These three algorithms are analyzed in [SHEK91].

CHAPTER 3

TRANSFORMATIONS

This chapter will examine six program transformations for nested set iterators like (3.1). Using analysis and examples, it will demonstrate that these transformations can reduce the execution time for a program.

In this thesis, we term the iteration through a set and its nested statements a **set loop**.

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

In query (3.1), the **for** D loop is a set loop that contains the statement `divisioncnt++` and another set loop. Because of the enclosed statements, the method of producing `Division|×|Employee` in (3.1) is more constrained than it would be in the relational setting—the join stream must be grouped by `Division`. We call loops like (3.1), 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 (3.1), query (3.1) would be a simple group-by loop.

Joins can be implicitly expressed with a group-by loop; they can also be expressed explicitly. The SQL query

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

can be expressed in *O++* [AGRA89] as the following **join loop**:

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

Ignoring output formatting, the two statements are equivalent.

3.1. 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. 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:

```
(3.4)  for (X1 of Set1) suchthat (Pred1(X1))
         ...
          for (Xm of Setm) suchthat (Predm(X1, ..., Xm))
              Sm1;
```

Since the order of iteration through each of the sets `Set1`, `Set2`, ..., and `Setm` is unspecified, after (3.4) has been executed, more than one program state may be reached (a program's state is determined by the values of variables, the output produced, etc...). Since the state reached may vary from program run to program run, statements like (3.4) make a program potentially non-deterministic.

Definition: The statement `Sm1` in (3.4) is **self-commutative relative to** `X1, X2, ..., and Xm` if (3.4) and

```
(3.5) for (X1 of Set1; ... ; Xm of Setm) suchthat
      (Pred1(X1) && ... && Predm(X1, ..., Xm))
            Sm1;
```

reach an identical, deterministic state from any starting state.

We will leave off the phrase **relative to** `X1, X2, ..., and Xm` 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.

This is a restrictive definition; few computations are likely to satisfy it in practice. Due to the finite precision of real arithmetic and the possibility of overflow errors, rearranging the loops may lead to a different result. Thus we provide two generalizations. A statement `Sm1` is **self-commutative ignoring overflow** if it would be self-commutative on a hypothetical machine where integer overflow errors cannot occur. A statement `Sm1` is **self-commutative ignoring finite precision** if it would be self-commutative on an infinite precision machine. Note that if `Sm1` is self-commutative ignoring overflow, it is also self-commutative ignoring finite precision, since integers can be represented as floating point numbers on an infinite precision machine. If `Sm1` is self-commutative ignoring overflow, and overflow does not occur at `Sm1` in either the transformed or the non-transformed program, the same value will be computed by `Sm1` in both. This is not true if `Sm1` 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.

3.1.1. Identifying Self-commutative statements

To identify members of the set of self-commutative statements, we make the observation that any given execution of a loop of the form (3.4) where none of the predicates have side-effects will produce the same program state as the execution of some loop-free code segment.[1] Conceptually, we are unrolling the loops on a per execution basis. We then have a large number of `Sml`-like statements that vary only in the values for `X1, ... ,Xm`. We will call these `Sml`-like statements **instantiations of** `Sml`. It may be that executing an arbitrary permutation of this sequence of instantiations will produce the same program state as (3.4) for a given execution. If this is true for all possible program executions, then `Sml` is self-commutative.

Self-commutative statements can be identified with a simple test. If any two adjacent instantiations of `Sml` can be permuted without changing the final computation of the program, `Sml` 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

---

[1] If `Professor` = { [Joe,1], [Jim,2], [Ralph,1] } and `Dept` = { [Math,1], [English,2] } and there are persistent pointers `P1-P3` to `Professor` elements and `D1-D2` to `Dept` elements, then the execution of

```
for (D of Dept)
   for (P of Professor) suchthat (D->did==P->did) {
      D->print(); P->print(); newline();
   }
```

that produces

```
Math 1 Joe 1
Math 1 Ralph 1
English 2 Jim 2
```

will produce the same program state as:

```
D1->print(); P1->print(); newline();
D1->print(); P3->print(); newline();
D2->print(); P2->print(); newline();
```

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

the statement sequence that increments `totpay` and `empcnt` is self-commutative ignoring overflow. (The sequence is self-commutative without qualification if no one has a negative salary—since then if any iteration order causes overflow to occur, all iteration orders do.) Since integer addition is associative and commutative, ignoring the effects of overflow, an arbitrary pair of instantiations of

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

like

```
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 takes place). We term statements such as those in (3.7) **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.[2] A more complete description of the class of self-commutative statements will be presented in Chapter 4.

3.1.2. 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 [KIM80] unless `Sm1`, the inner statement, is self-commutative. To see this, suppose that blocked-nested-loops is used to execute

---

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

[3]Examples of non-determinism resulting from non-self-commutatative statements can be found in Sections 3.2.3.1 and 4.1.

```
(3.8) for (D of Dept)
        for (P of Professor) suchthat (D->did==P->did) {
           printf("%s %d %s",D->name,D->did,P->name);
           newline();
        }
```

under the assumption that two objects fit on a page and two buffer pages are available. If `Professor` = { [Joe,1], [Jim,2], [Ralph,1] }, `Dept` = { [Math,1], [English,2] }, and (3.8) is executed using a blocked-nested-loops algorithm, the output sequence

Math 1 Joe
English 2 Jim
Math 1 Ralph

is produced. However, this violates the **group by** semantics of (3.8). Most other join algorithms including **Hybrid-hash**, index-nested-loops, and sort-merge will, however, produce legal groupings. These algorithms find all `Professors` that join with the first `Dept`, then all `Professors` that join with the second `Dept`, and so on. Thus, they group the join stream by `Dept`.

3.2. OPTIMIZATIONS

As illustrated by (3.8), nested iterators can be used to express a join with a grouping constraint. These grouping constraints can hurt 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 (3.8) to use elements of a temporary set instead of elements of `Dept` and `Professor`) to enable more reorderings to be made without modifying the program's semantics. Throughout this section, we will assume that predicates are side-effect free.

3.2.1. Simple Group-by Loops

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

```
(3.9) for (X1 of Set1) suchthat (Pred1(X1))
        ...
          for (Xm of Setm) suchthat (Predm(X1, ..., Xm))
              Sm1;
```

If `Sm1` is a self-commutative statement, then, by the definition of self-commutativity, (3.9) is equivalent to:

```
(T1) for (X1 of Set1; ... ; Xm of Setm) suchthat
     (Pred1(X1) && ... && Predm(X1, ..., Xm))
          Sm1;
```

Even if Sm1 is not self-commutative, if Sm1 does not modify Set1-Setm or the variables used in the predicates, (3.9) 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 of Temp) /* in the sorted order */
         Sm1';
```

In transformation (T2), the Temp set loop contains a statement Sm1' that looks like Sm1, 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(Xi) refers to the fields of Seti that are used in statement Sm1 or that are needed for sorting. It includes a unique identifier for each Xi object other than Xm for use in sorting; if the user has not supplied a primary key, Needed(Xi) includes the object identifier (oid) of Xi as the identifier. The asymmetric treatment of Xm in transformation (T2) allows the transformed program to maintain the proper semantics while minimizing cost. Statement (3.9) 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 X1 varies the most slowly, followed by X2, X3, ... Sorting on the composite key maintains something slightly stronger than this semantic requirement.[4] The transformed program will behave as if it was iterating through Seti $\forall i\ 1 \le i \le (m-1)$ in the same order each time, although the transformed program may behave as if it was iterating through Setm in a different order each time. The asymmetric treatment maintains the program semantics without wasting space in each Temp object for a unique identifier for the relevant Xm object.

3.2.1.1. 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. Sm1 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

[4]Hashing could also be used for grouping. Hashing on a unique identifier for X1 can be used to partition (group) elements by X1 values. Each partition can be sorted individually, or further hashing can be used.

that each join operator is an equijoin. We further assume that all set elements are clustered together on disk. We will define two functions for use in the analysis. The first is $\delta(s,p)$, the number of objects of size s that fit on a page of size p:

$$\delta(s,p) = \left\lfloor \frac{p}{s} \right\rfloor$$

The second is $\theta(m,s,p)$, the number of pages of size p required to hold m objects of size s:

$$\theta(m,s,p) = \left\lceil \frac{m}{\delta(s,p)} \right\rceil$$

To simplify the following analysis, we only analyze the I/O costs in this chapter. Since the amount of CPU time required by the **Hybrid-hash** join algorithm (the algorithm used in all the examples in this chapter) is roughly proportional to the number of I/Os performed, we do not believe that incorporating CPU costs would significantly change the results obtained.[5] The notation in Table 3.1 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 of set S
sel_S	predicate selectivity for set S
O_S	number of objects per page, $O_S = \delta(size_S, P)$
P_S	number of pages, $P_S = \theta(\|S\|, size_S, P)$
F	size of a main memory hash table for S is $F \cdot P_S$ (i.e. F captures overhead)

Table 3.1

[5]CPU cost is a minor fraction of the total cost of the algorithms analyzed in this thesis. However, the cost functions for our implementation include CPU cost. We used System R [SELI79] style optimization—so the cost function was produced using a combination of the number of pages read and the number of tuples requested from the storage manager. This is necessary to get an accurate estimate of the cost of the nested-loops join algorithm.

In this chapter, we will assume that queries are evaluated using the centralized **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 N partitions R_1 R_2, ..., and R_N, on the basis of a hash function applied to the join attribute. R_1 is kept in a main memory hash table; the R_j are written to disk $\forall j\ 2 \le j \le N$.

(2) Select and partition S in the same way; this implies that R_i will only join with S_i $\forall i\ 1 \le i \le N$. The value of N is chosen so that: (1) S_1 can be joined in memory with R_1 as S is being partitioned, and (2) the hash table for R_j can fit in memory $\forall j\ 2 \le j \le N$.

(3) After S has been partitioned and S_1 has been joined with R_1, each R_j is joined as follows with S_j $\forall j\ 2 \le j \le N$. R_j is read into memory, and each R_j object is hashed on its join value and inserted into a hash table using a new hash function. S_j is read into memory a page at a time, and the join attribute of each S_j object is used to probe the hash table to find the R_j objects that it joins with.

Thus, the join is broken into N smaller joins. If the hash function produces an R_i $\exists i\ 1 \le i \le N$ partition that will not fit in the available memory, an overflow tactic like Simple Hash Join [DEWI84] must be employed. This is an important complication in an actual implementation, but it is a peripheral issue to our discussion of how to optimize a DBPL, so this thesis will not discuss it further or include it in the examples.

[DEWI84,SHAP86] provide the following analysis of **Hybrid-hash**'s performance that we extend to include the effects of selection and projection. Let $P_{Hash} = (P_{R'} \cdot F)$, the size of a hash table for all of R′. Then, $B = \left\lceil \frac{P_{Hash} - M}{M-1} \right\rceil$ is the number of partitions of R′ that will need to be written to disk. Since $(M-B)$ is the amount of space available for the hash table for R_1, $q = \min(1.0, \frac{M-B}{P_{Hash}})$ is the fraction of R′ contained in partition R_1. (The min is required because q must be no larger than 1.0.) Finally, $N = (B+1)$ since one partition, R_1, is kept in main memory. 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,[6] 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 of S (or S_1)—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 (i.e. R).

[6]The partitions are written randomly; all the reads are sequential.

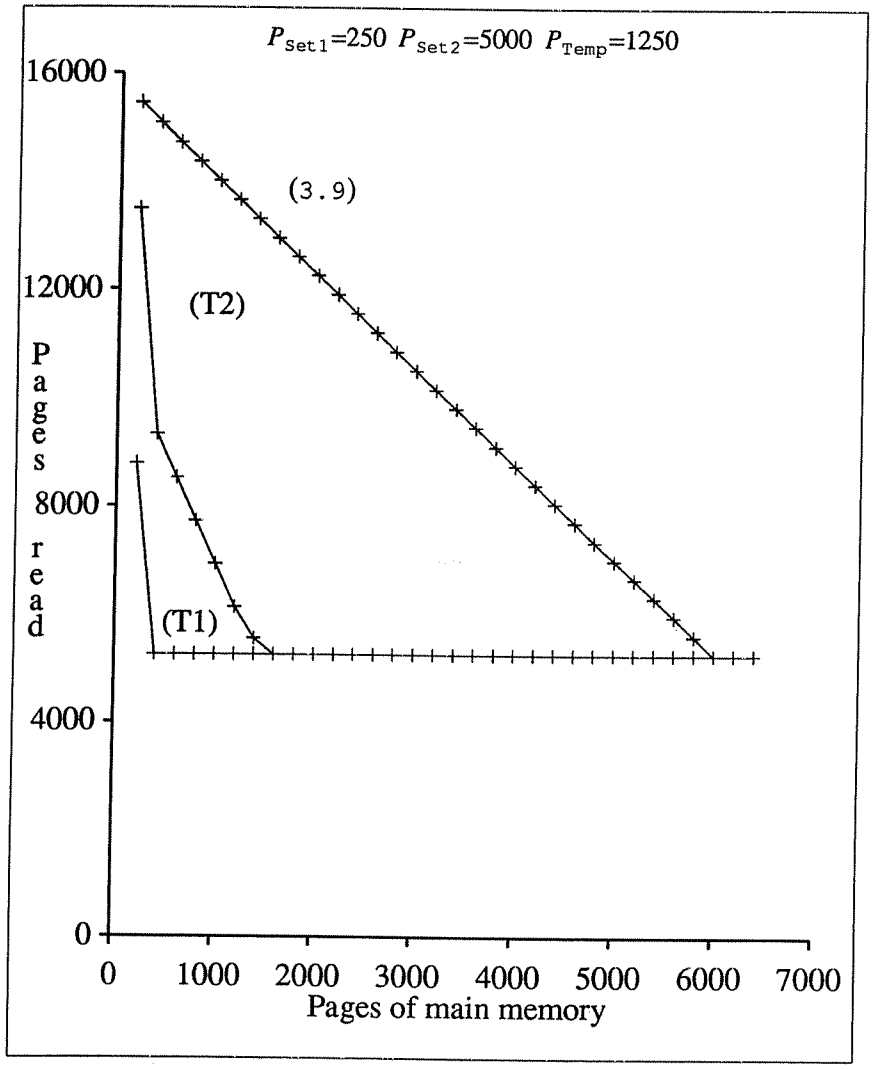

Figure 3.1

Here, the concept of self-commutativity becomes useful because the optimizer cannot choose the probing set in (3.9) (as (T1) allows) unless the statement Sm1 is freely-permutable. Figure 3.1 compares the performance predicted by the analysis of the untransformed simple group-by loop (3.9) 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 entirely 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 permits the use of Set2 as 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 buffer Temp objects). By our hypothesis that each element of the smaller set, SmallSet, that satisfies the selection criterion joins with exactly k elements of the larger set, |Temp| = (|SmallSet| · $sel_{\texttt{SmallSet}}$ · k). Note that $size_{\texttt{Temp}} = (\pi width_{\texttt{Set1}} + \pi width_{\texttt{Set2}})$. Given this, $P_{\texttt{Temp}}$ can be calculated using the formula in Table 3.1. Assume that $M > \sqrt{P_{\texttt{Temp}}}$. Then Temp can be sorted in two passes [SHAP86], and the cost to execute the query resulting from transformation (T2) is

join cost of Set2|×|Set1 with $M_1 = (M\text{-}1)$ buffer pages	Join using Set2 as the outer set
+Unbuff($P_{\texttt{Temp}}$)	Write pages of Temp that cannot be buffered during join (see definition below)
+Unbuff($P_{\texttt{Temp}}$)	Read pages of Temp written during join for first pass of sort
$+P_{\texttt{Temp}} \cdot 2$	Write runs, and read them back for sort
$-\min(P_{\texttt{Temp}}, M-1-\sqrt{P_{\texttt{Temp}}}) \cdot 2$	I/O savings if extra memory available during sort [SHAP86]

The memory resident hash table for Set1 has first claim on memory during the join, so if $(P_{\texttt{Set1}'} \cdot F) \geq M_1$, 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 $(M_1 - P_{\texttt{Set1}'} \cdot F)$ pages available for buffering, so only $(P_{\texttt{Temp}} - (M_1 - P_{\texttt{Set1}'} \cdot F))$ 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 $\max(0, (P_{\texttt{Temp}} + P_{\texttt{Set1}'} \cdot F - M_1))$. Thus,

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

In Figure 3.1, we graph the performance of queries (3.9) and (T1) against the performance of the query produced by transformation (T2), assuming that $P_{\texttt{Temp}} = 1250$. 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 (3.9) and (T1). Transformation (T2) is more expensive than (T1) because it must sort Temp. (T2) performs better than (3.9) because it uses Set2 as the outer set—sorting Temp is cheaper than using Set2 as the inner set (because Temp is one fourth the length of Set2). Thus, transformation (T2) can be an attractive option for executing queries like those in (3.9) when the statement sequence Sm1 is not self-commutative.

3.2.2. General Group-by Loops

The group-by loops exemplified by (3.9) 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 query

```
(3.10) for (X1 of Set1) suchthat (Pred1(X1)) {
         S11;
         for (X2 of Set2) suchthat (Pred2(X1,X2))
             S21;
         S12;
       }
```

exemplifies the general case since any of the statements S11, S21, or S12 may contain a set loop.

In the following sections, unless stated otherwise, we will assume that none of the statements S11, S21, or S12 insert (delete) elements into (from) Set1 or Set2. We further assume that S11 and S12 do not change the values of elements of Set2 or the values of elements of Set1 other than the current X1. The transformations presented are general—they can be used with a variety of join methods. However, for uniformity in this chapter's analysis, we will assume that **Hybrid-hash** is used to evaluate all joins.

3.2.3. 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, (3.10) cannot be rewritten unless the conditions in Section 3.2.3.3 are met. The following is an example of interference:

```
(3.11) for (D of Dept)  {
         cnt = 0;  //S11
         for (P of Professor) suchthat (D->did==P->did)
               cnt++;  //S21
         printf("%s %d", D->name, cnt); newline(); //S12
       }
```

Since the definitions of cnt in S11 and S21 both reach the use of cnt in the printf statement, the two

loops interfere. Example (3.11) can be rewritten using the technique described in Section 3.2.3.3. 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 (e.g. Professor in (3.11)) must be the inner set for the join.

3.2.3.1. 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, (3.10) can be rewritten as:

```
(T3) for (X1 of Set1) suchthat (Pred1(X1)) {
        S11; S12;
     }

     for (X1 of Set1) suchthat (Pred1(X1))
        for (X2 of Set2) suchthat (Pred2(X1,X2))
            S21;
```

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 S11;S12 is self-commutative relative to X1; or (3) S21 is self-commutative relative to X1 and X2. Requiring that one of these conditions hold is necessary to avoid subtle forms of inconsistency. Consider:

```
(3.12) t1 = new Tree; t2 = new Tree;  //create empty trees
       for (X1 of Set1) {
          t1->Insert(X1->i);          //Not self-commutative
          for (X2 of Set2)
              t2->Insert(X1->i);     //Not self-commutative
       }
```

Suppose Set2 has exactly one element. Then, since there are no predicates, (3.12) will produce the same program state as:

```
t1 = new Tree; t2 = new Tree;
for (X1 of Set1) {
   t1->Insert(X1->i);
   t2->Insert(X1->i);
}
```

However, if (3.12) is transformed using (T3) to:

```
(3.13) t1 = new Tree; t2 = new Tree;
       for (X1 of Set1)  //iterate in order {[1],[2]}
          t1->Insert(X1->i);

       for (X1 of Set1)  //iterate in order {[2],[1]}
          for (X2 of Set2)
             t2->Insert(X1->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 (3.13), `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 (very) 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, such a 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. Since there is no flow of values, the only possible interaction between the inner and outer loop is through a constraint that a value computed in the outer loop have some relationship to one computed in the inner loop. Such a constraint will only inhibit transformation 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 in practice, so that transformation (T3) can be applied. The group-by loop produced by (T3) may be optimized using the techniques described in Section 3.2.1.

Analysis and Example of Flow Dependence Free Optimization

Transformation (T3) is beneficial if `S11` or `S12` chase disk pointers or iterate through a set since moving them out of the original loop reduces contention for buffer pages during the computation of `Set1` |×| `Set2`. However, we will only analyze the case where transformation (T3) exposes an opportunity for (T1) to be applied 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:

join cost of `Set2 |X| Set1`	Join with `Set2` as the outer set.
	Combine scanning and partitioning of `Set1`.

Note that if the outer loop had been a join loop instead of a set scan, this analysis is too simplistic. For instance, the application of (T3) and a loop interchange could produce:

```
(3.14) for (A of SetA; B of SetB) suchthat (Pred1(A,B)) {
          S11;
          S12;
       }
       for (X2 of Set2) suchthat (Pred2(X1,X2))
          for (A of SetA; B of SetB) suchthat (Pred1(A,B))
             S21;
```

In (3.14), it is likely that all the buffer pages will be needed for computing `SetA|X|SetB`. Thus, the scanning and partitioning will probably have to be done in separate passes. A formula that covered join loops would have to include both the cost of producing the join `SetA|X|SetB` and of writing the result of the join to disk so that it could be used by the second set loop in (3.14). Similar comments can be made about future transformations where the analysis assumes that scanning and partitioning are performed simultaneously.[7]

As an example of transformation (T3), consider:

```
(3.15) for (D of Dept) {
          Univbudget += D->budget;
          for (P of Professor) suchthat (D->did==P->did)
          {
             cnt[D->floor]++;
               /* Calculate number of profs/floor */
             if( D->majors > (TotalMajors/5) )
                 High_Major_dept_sals += P->salary
          }
       }
```

Note that the statement sequence in the `Professor` loop computes two aggregates. Using (T3), query (3.15) can be transformed to the following (skipping a few intermediate steps):

---

[7]In the current implementation described in Chapter 5, the code generated takes advantage of data being in the buffer pool. However, the cost formulas assume that `Set1` (or `SetA|X|SetB`) must be read for both loops, since the cost of each loop is computed separately and then the two costs are added. This may cause a non-optimal plan to be chosen; however, since the cost of a scan is small compared to the cost of a join, a plan very close to optimal will be chosen even if the optimal plan is not. This was the easiest way to ensure that the cost functions for loops like (3.14) are reasonably estimated. Thus, the number of I/Os performed (and, hence, the execution time) in the implementation is what would be expected on the basis of the cost formulas in this chapter, while the optimizer will slightly overestimate the number of I/Os performed in some cases.

```
(3.16) for (D of Dept)
          Univbudget += D->budget;

       for (P of Professor)
          for (D of Dept) suchthat (D->did==P->did) {
              cnt[D->floor]++;
              if( D->majors > (TotalMajors/5) )
                  High_Major_dept_sals += P->salary
          }
```

Hybrid-hash can be used for both (3.15) and (3.16). This is clear for (3.16), but (3.15)'s implementation is somewhat more complex. While (3.15) scans the partitions of `Dept`, `Univbudget` is incremented for each `Dept` object before probing the `Professor` hash table with that object (e.g. an outerjoin is used instead of a join).

Suppose that `Dept` consists of 50 pages, that `Professor` 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 3.2.1.1, the set loop in (3.15) will cost 2633 page reads. However, the join in (3.16) will only cost 1050 page reads, since a hash table for all of `Dept` will fit in main memory. We simultaneously scan `Dept` for the first set loop while loading it into a hash table for the join loop, so the total cost of (3.16) 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 `UniversityTotalMajors()` is an in-line function to calculate the total number of declared majors in all `Depts`. Then the following code fragment

```
TotalMajors = UniversityTotalMajors();
   //Function uses the global Dept set
for (D of Dept) //from (3.16)
   Univbudget += D->budget;
```

can be expanded to

```
(3.17) Dept * D1;
         //Declare D1 to be a pointer to a Dept object
       TotalMajors = 0;
       for (D1 of Dept)
          TotalMajors += D1->majors;
       for (D of Dept)
          Univbudget += D->budget;
```

Simple multi-query analysis allows (3.17) to be rewritten as:

```
TotalMajors = 0;
for (D of Dept) {
   TotalMajors += D->majors;
   Univbudget += D->budget;
}
```

Since `TotalMajors` is used in the inner loop of (3.15), the evaluation of `UniversityTotalMajors()` could not have been pushed inside the `Dept` loop of (3.15). 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.

3.2.3.2. 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 (3.10), however, values computed in the outer loop will often be used in the inner. (We repeat (3.10) below as (3.18) for ease of exposition.)

```
(3.18) for (X1 of Set1) suchthat (Pred1(X1)) {
          S11;
          for (X2 of Set2) suchthat (Pred2(X1,X2))
             S21;
          S12;
       }
```

If no values flow from `S21` to the outer loop, the loops do not interfere, and `S21` does not modify any elements of `Set1`; but values do flow from `S11` or `S12` into the inner loop, a somewhat more complicated transformation than (T3) is required. Let `v1, ... ,vt` be the variables written by `S11` and `S12` that are used by expressions in the inner loop. Then (3.18) can be rewritten as:

```
(T4) Temp = []; //empty sequence
     for (X1 of Set1) suchthat (Pred1(X1)) {
        S11;
        Append <Needed(X1), v1, ... ,vt> to Temp.
        S12;
     }
     for (T of Temp)  /* in insertion order */
        for (X2 of Set2) suchthat (Pred2'(T,X2))
           S21';
```

`S21'` is `S21` rewritten to use fields of `Temp` instead of `vi` $\forall i\ 1 \le i \le t$ and instead of the fields of `Set1`. `Pred2'(T,X2)` is a similarly rewritten version of `Pred2(X1,X2)`. `Needed(X1)` contains the fields of `X1`

used in `Pred2(X1,X2)` and `S21`. Note that if `Pred1(X1)` is very restrictive or objects in `Temp` are smaller 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 `Set1` 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, `S11` or `S12` 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`. Note also that if `S21'` is self-commutative, `Temp` may be a set.

Analysis and Example of Optimization in the Presence of Flow Dependences

Although transformation (T4) is appropriate under several conditions, we will consider only the case where `S21'` is self-commutative. In this case, the two loops can be flipped. Note that $size_{\texttt{Temp}} = (\pi width_{\texttt{Set1}} + \sum_{i=1}^{n} \texttt{sizeof(vi)})$, and that |Temp| = (|Set1| · $sel_{\texttt{Set1}}$). Once again, if we can combine scanning and partitioning, the overall cost of (T4) will be:

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

As an example of a case where rewrite (T4) is profitable, consider a slightly modified version of the previous example in which a `Dept`'s `floor` 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`.

```
(3.19) for (D of Dept) {
          floor = D->buildinginfo->floor; //S11
          for (P of Professor) suchthat (D->did==P->did)
             cnt[floor]++; //S21
       }
```

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 (3.19) as:

```
(3.20) Temp = {}; //S21 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 (T->did==P->did)
             cnt[T->floor]++;
```

Moving `D->buildinginfo->floor` into the loop and eliminating the assignment to `floor` is another alternative; this alternative produces a group-by loop with a self-commutative statement inside. The loops can then be interchanged to produce:

```
(3.21) for (P of Professor)
          for (D of Dept) suchthat (D->did==P->did)
             cnt[D->buildinginfo->floor]++;
```

Assume that there are 300 buffer pages, that `Dept` consists of 50 pages and 500 objects, and that `Professor` consists of 1000 pages and 10,000 objects. Suppose that the objects that `buildinginfo` points to are rather large (so that very few fit on a page) and that only one page is allocated for pointer dereferencing. Then, every time `buildinginfo->floor` is dereferenced, one page read is performed. Given these assumptions, the original query (3.19) will incur 2633 page reads for the join and another 500 page reads for the dereferencing operations. Query (3.20) costs 50 I/Os to read `Dept` and 500 I/Os for the dereferencing operations. Since $P_{\texttt{Temp}}$ is so small, `Temp` can be buffered in main memory, so the cost of the join will only be the 1000 I/Os to read the `Professor` set. Thus, the total cost of query (3.20) is 1550 I/Os. Alternative (3.21) is the worst plan; its dereferencing operations alone require 10,000 I/Os (assuming each `Professor` joins with exactly one `Dept`).

3.2.3.3. Loops Used As Aggregates On Grouped Values

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

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

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

```
(3.22) for (D of Dept) {
          cnt = 0;  //S11
          for (P of Professor) suchthat (D->did==P->did)
             cnt++;  //S21
          printf("%s %d", D->name, cnt); newline(); //S12
       }
```

We consider a transformation from [SHOP80] to rewrite queries involving aggregate functions such as (3.22). (We repeat (3.10) as (3.23) for ease of exposition.) In

```
(3.23) for (X1 of Set1) suchthat (Pred1(X1)) {
          S11;
          for (X2 of Set2) suchthat (Pred2(X1,X2))
             S21;
          S12;
       }
```

suppose that S11 can be partitioned into two sets of statements: those whose values flow only to S11 and to outside the **for** X1 loop and those that assign constants to variables v1, ..., vt.[8] The v1, ..., vt must be assigned to during each pass through S11. In S21, they may be employed only in reduction operations; in S12, they may be read but not written. Statements in S12 may only have values flow back to S12 and to outside the **for** X1 loop. S21 and S12 must not modify any elements of Set1. Finally, Set2 must not be nested inside an object of Set1. If these conditions are met, (3.23) can be rewritten as:

```
(T5) Temp = [];
     for (X1 of Set1) suchthat (Pred1(X1)) {
        S11;
        Insert <Needed(X1), v1, ..., vt> into Temp;
     }
     for (X2 of Set2)
        for (T of Temp) suchthat (Pred2(T,X2))
           S21';
     for (T of Temp)
        S12';
```

provided S21′ is self-commutative relative to X2 and T. Needed(X1) are the fields of Set1 mentioned in S21, S12, or Pred2(X1,X2). S21′ and S12′ are rewritten versions of S21 and S12 that replace uses and definitions of vi with uses and definitions of T->vi. They also replace uses of fields of Set1 with uses of the fields of Temp (i.e. X1->a is replaced with T->a). If S11 or S12 is self-commutative, Temp can be a set provided that Set1 is an explicit set and not the implicit set resulting from a join.

---

[8]Actually, **commutative assignments** to variables v1, ..., vt are allowed. The phrase **commutative assignment** will be defined in Chapter 4.

Analysis and Example of Optimizing Aggregate Computation

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

$P_{\texttt{Set1}}$	Scan of `Set1`
+join cost of `Set2`|×|`Temp`	Join with `Set2` as outer set
$-P_{\texttt{Temp}}$	Combine partitioning of `Temp` with scan of `Set1`
$+\begin{cases} 2 \cdot P_{\texttt{Temp}} & \text{if Temp's hash table does not fit in main memory} \\ 0 & \text{otherwise} \end{cases}$	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 (3.22) can be rewritten as:

```
(3.24) Temp = {};
       for (D of Dept) {
          cnt = 0;  //S11
          Insert <D->did, D->name, cnt> into Temp;
       }
       for (P of Professor)
          for (T of Temp) suchthat (T->did==P->did)
             T->cnt++; //S21'

       for (T of Temp) {
          printf("%s %d", T->name, T->cnt);
          newline(); //S12'
       }
```

Using the same parameters as before, the unmodified query (3.22) performs 2633 I/Os. The rewritten query (3.24) requires 50 reads to scan `Dept`. `Temp` is inserted in a memory resident hash table during this scan. Thus, the join does 1000 reads, since only `Professor` 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 (3.24) is 1050 I/Os.

3.2.3.4. An Analogue to Relational Join Associativity

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

```
(3.25) for (X1 of Set1) suchthat (Pred1(X1)) {
          S11;
          for (X2 of Set2) suchthat (Pred2(X1,X2)) {
             S21;
             for (X3 of Set3) suchthat (Pred3(X1,X2,X3))
                S31;
             S22;
          }
          S12;
       }
```

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 (3.25) provided two conditions are met: (1) (3.25) does 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 (3.25), conditions on the flow of values must be met. Thus, the associativity rewrite will sometimes be applicable when the others are not. In the rewrite, `last` is the element in `Temp` before the current element being examined (i.e. `T`), and `next` is the element immediately after. All the fields of `last` will be NULL when `T` is the first element of `Temp`; all the fields of `next` will be NULL when `T` is the last element of `Temp`. The associativity rewrite is the following:[9]

[9](T6) in [LIEU91] was incorrect, so this (T6) differs from the one there.

```
(T6) Temp = []; /* Empty sequence */
     for (X2 of Set2) suchthat (Pred2'(X2)) {
        for (X3 of Set3) suchthat (Pred3'(X2,X3))
           Append <oid(X2),Needed(X2),Needed(X3)>
              to Temp;
        if(nothing was added to Temp in for X3 loop)
           Append <oid(X2),Needed(X2),NULL> to Temp;
             /* Doing an outer join */
     } //produces Set2|X|_Set3, the left outerjoin

     for (X1 of Set1) suchthat (Pred1(X1)) {
        S11;
        for (T of Temp) suchthat (Pred2''(X1,X2)) {
        // in insertion order
           if (last->oid != T->oid)
               S21'';
               //Do S21'' if there is a new X2 value
           if (T does not have NULL for Needed(X3)
                     && Pred3''(X1,X2,X3))
               S31''; //Do S31'' if there is a non-NULL
                      // X3 value && Pred3''
           if (next->oid != T->oid)
               S22'';
               //Do S22'' if this is the last element
               //corresponding to an X2 value
        }
        S12;
     }
```

`Needed(X2)` are the fields of `Set2` mentioned in `S21`, `S31`, or `S22` or used in `Pred2(X1,X2)` or `Pred3(X1,X2,X3)`. The definition is similar for `Needed(X3)`. `Pred2'(X2)` and `Pred3'(X2,X3)` are derived from `Pred2(X1,X2)` and `Pred3(X1,X2,X3)` by making them less restrictive. All clauses of the predicate that involve either `X1` or variables written by (3.25) 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 `T->a` instead of `X2->a` and `T->b` instead of `X3->b`. Parts of the predicate `Pred3(X1,X2,X3)` that refer only to `Set2`, `Set3`, and variables constant for the duration of the loop are removed from `Pred3''(X1,X2,X3)`, since they are already ensured to be satisfied by `Pred3'(X2,X3)` in the first loop.

We would like a simple transformation like `(Set1 |X| Set2) |X| Set3` → `Set1 |X| (Set2 |X| Set3)` where the left-hand side corresponds to (3.25). However, since `S21` and `S22` must be executed for every qualifying object of `Set2`, care must be taken. Ensuring that `S21` and `S22` are executed the proper number of times

requires that no qualifying Set2 objects are lost. As a result, an outerjoin must be used. If S21 and S22 are null statement sequences, the check for NULL can be removed, the outerjoin can be replaced with a join, and oid(X2) will not need to be stored in Temp objects. A sequence is used to ensure that S21, S31, and S22 are executed in the same order as they would have been in the unmodified query (3.25) if Set2 and Set3 were sequences. Since Temp is a sequence, objects of Temp will be partitioned for **Hybrid-hash** in insertion order. By maintaining the scan order within the partition—which will be done in a straightforward implementation of **Hybrid-hash**—the Set1|×|Temp loop in (T6) will produce a join stream that is properly grouped.

Analysis and Example of Associativity Rewrite

In this rewrite, $size_{\texttt{Temp}}$ = (sizeof(oid) + $\pi width_{\texttt{Set2}}$+$\pi width_{\texttt{Set3}}$).[10] Assuming once again that exactly $k \geq 1$ elements of the larger set join with each element of the smaller set (SmallSet) that passes the join criterion, |Temp| = (|SmallSet| · $sel_{\texttt{SmallSet}}$ · 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
+$P_{\texttt{Temp}}$	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 professors (and their specialties) by department:

```
(3.26) for (D of Dept) {
         D->print(); //S11
         for (P of Professor) suchthat (D->did==P->did)
            for (S of Specialty) suchthat
            (S->id==P->specialty) {
               P->print();
               printf(" %s ", S->title);
               newline(); //S31
            }
       }
```

which can be rewritten as:

[10]The implementation described in Chapter 5 replaces the use of an oid with an integer counter, so $size_{\texttt{Temp}}$ is smaller there than described here. The exposition was simpler here using an oid.

```
(3.27) Temp = [];
       for (P of Professor) {
          for (S of Specialty) suchthat
          (S->id==P->specialty) {
             Append <Needed(P), S->title> to Temp;
          }
         if (No Specialty joined with P->specialty)
            Append <Needed(P), NULL> to Temp;
       }
       for (D of Dept) {
          D->print(); //S11
          for (T of Temp) suchthat (D->did==T->did)
             if (T->title != NULL) {
                T->print();
                printf(" %s ", T->title);
                newline(); //S31'
             }
       }
```

Note that in (3.27), since S21 and S22 are empty, the outerjoin and the check for NULL could have been eliminated. We included them for illustrative purposes. Suppose Dept has 10 pages, Professor has 100 pages, Specialty has 20 pages, F=1.2, $size_{\texttt{Dept}} = 2 \cdot size_{\texttt{Professor}}$, and the buffer pool has 31 pages. For (3.26), Dept|X|Professor 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 Professor joins with exactly one Dept. Since each Professor and Dept is printed in its entirety, the objects resulting from this join will be three times longer than Professor objects, so 300 pages must be written to disk. We then join this 300 page result with Specialty. Since Specialty will fit in a main memory hash table, this join will only require reading the Specialty and the results of the Dept|X|Professor join once— so only 320 I/Os will be needed. The total cost of (3.26) is 903 I/Os.

For (3.27), since only the title is needed from Specialty, we will assume that $size_{\texttt{Temp}}$ is about 20% longer than $size_{\texttt{Professor}}$. Assuming each professor has one specialty, $P_{\texttt{Temp}}$ will be 120 pages. Specialty 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 Dept|X|Temp will cost 344 I/Os. Thus (3.27) will cost 584 I/Os.

3.2.3.5. Handling Pointers

To take full advantage of our techniques, queries like (3.19) that chase disk pointers should be rewritten as joins if possible by using pointers as join attributes. Such rewrites are only possible if the system knows which pages are referenced by the pointer chasing—for instance, if an **extent**, a set of all the objects of a particular type,

exists.[11] Suppose there is an extent of all `Buildings`. Then

```
for (D of Dept) {
    floor = D->buildinginfo->floor; //S11
    for (P of Professor) suchthat (D->did==P->did)
        cnt[floor]++; //S21
}
```

can be rewritten as

```
for (D of Dept) {
    for (B of Building) suchthat (D->buildinginfo==B)
        for (P of Professor) suchthat (D->did==P->did)
            cnt[B->floor]++; //S21
}
```

Our transformations and analysis can be directly applied to such a loop. This sort of preprocessing should be applied to loops before our transformations are applied.

3.3. SUMMARY

Processing joins expressed as nested set loops can be expensive. In this chapter, we presented six transformations that can be applied to a group-by loop query to produce an equivalent query with fewer constraints on how it executes. These transformations employ extra set scans, temporary sets, set sorting, and rewriting statements enclosed in set loops. The conditions under which the transformations can safely be applied are described using dataflow analysis.

We also defined and made extensive use of 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. This analysis was foundational to making the transformations useful. The most basic transformation, (T1), was based on this observation.

In addition to presenting the optimizations, we analyzed the performance of queries before and after transformation. We used the analysis and examples to demonstrate that this chapter's transformations can improve a program's execution time.

---

[11]The **Find-children** algorithms of Chapter 6 can be used to calculate the referenced pages as well.

CHAPTER 4

SELF-COMMUTATIVITY

The notion of self-commutativity was extensively used in Chapter 3. However, the class of self-commutative statements was never carefully characterized. We will now examine the class.

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

```
(4.1) for (X1 of Set1) suchthat (Pred1(X1))
        ...
          for (Xm of Setm) suchthat (Predm(X1, ..., Xm))
              Sml;
```

If (4.1) and

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

will produce an identical, deterministic result from any starting state, then `Sml` is self-commutative.

We will assume in this chapter that `Sml` does not modify sets `Set1-Setm` except perhaps by performing reduction operations on (or assignments of expressions that are constant for the duration of the loop to) fields of `X1-Xm`. We will also assume that the predicates do not have side-effects or use variables or fields of iterator variables that are modified by `Sml`. We will characterize a subclass of self-commutative statements in Sections 4.1-4.3. In Section 4.4, we will prove the correctness of our characterization.

4.1. SELF-COMMUTATIVE ARITHMETIC OPERATIONS

We begin the identification of **self-commutative** statements by first considering the case where `Sml` is a single statement that performs numeric computation. Throughout this thesis 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 `Sml` is

```
(4.2) v op= f(X1, ... ,Xm,v1, ... ,vt);
```

This statement is self-commutative ignoring overflow if the `vi` are not modified by `Sml` (i.e. `v` is not one of the `vi`), `f` is a mathematical function (an expression that produces a value and has no side-effects), `v` is an integer variable that is not used in any loop predicates in (4.1), and `f` does not examine any fields of `X1-Xm` that are

modified by Sm1. To see this, consider an example when the nesting is only one level deep (i.e. m=2):

```
for (D of Division)
   for (E of Employee) suchthat (D==E->division)
      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. The same principle holds for an arbitrary statement of the form (4.2) 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 (4.2) **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 (4.2) 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.)

The "variable" v may also be a field of an iterator variable Xi->fj provided Xi->fj is not used by a loop predicate or examined by any statements in Sm1 except implicitly by the reduction operation. We term a modified field that meets these properties a **delta-field**. In the discussion, we ignore the detection of aliases. In practice, a conservative dataflow analysis that finds all potential aliases must be used. If two variables (or delta-fields) cannot be proved to reference different locations, they must be treated as being the same.

Now, consider Sm1 when it contains several statements that perform numeric computations. The + and - operators for integers can be treated as inverses ignoring overflow. (The * and / operators are not inverses for integers since $0 = (1/3)*4 \neq (1*4)/3 = 1$.) Inverses are important because Sm1 may contain several statements of the form (4.2) 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 (4.2). For instance, the statement sequence `v+=E1; v-=E2` is equivalent to the single statement `v+=(E1)-(E2)`, which is of form (4.2). However, the statement sequence `v+=E1; v*=E2` is not equivalent to any statement of the form (4.2).

The +=, -=, *=, and /= operators for reals of the form (4.2) (where `v` is now a real variable/delta-field 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(X1, ... ,Xm,v1, ... ,vt)` where `f` is a mathematical function that does not examine fields of `X1-Xm` modified by `Sm1` and the `vi` are either **temporaries** or variables not modified by the loop. A **temporary** is a variable that must be defined by a commutative assignment before any use in `Sm1`. Also, no assignment to a temporary may reach any use outside of `Sm1`.[1] Since temporaries are always defined before they are used in `Sm1` and only the current iteration of the loop uses the value, **commutative assignments** to temporaries are always self-commutative.

We can also loosen our characterization of a reduction. The `vi` in (4.2) can either be variables constant for the duration of the set loop or temporaries. Since the original characterization of (4.2) 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,

```
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 (4.2). The presence of conditional statements makes the equivalent macro expression unintuitive, but our restrictions on predicates in Section 4.3 will ensure that each use of a temporary is equivalent to the use

[1] 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. This is not a matter of correctness because of the non-deterministic iteration order semantics of **`for`** loops. However, since self-commutative statements must produce a deterministic result, such statements are not self-commutative.

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. Since any loop with commutative assignments can be rewritten as an equivalent loop without them (though the equivalent loop may be less efficient than the original loop), we will ignore them in our proof in Section 4.4.

The class of self-commutative statements also includes statements of the form (4.2) 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.

The main intuition for the rules is that making explicit use of a variable (or delta-field) defined by a reduction makes a statement sequence non-self-commutative. For example, `v += v + f(X1)` and `{v += f(X1); r += v;}` are not self-commutative. These statements are admittedly strange—the first is equivalent to `v = v + v + f(X1)`. The rules eliminate them 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 `X1` from previous iterations of the loop to be implicitly reused during the computation made during the current iteration of the loop—which makes iteration order significant. For instance, in

```
v = 0;
for (X1 of Set1)   //Set1 = {[1],[2]}
   v += v + X1->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 makes explicit use of `v` in `Sm1`, that statement is not self-commutative.

Besides reductions, the assignment of an expression constant for the duration of the loop to a variable/delta-field `v` is self-commutative provided that no other statements in `Sm1` modify `v` or examine its value. Such a statement can only have a flow of values to outside the loop. It doesn't matter at what point the assignment is made provided it is performed before the loop ends. Thus, such a statement is self-commutative. These statements can be viewed as reductions. A reduction `v op= E` involves a function `op` such that `op(op(v,E1),E2) = op(op(v,E2),E1)`. The assignment `v=C` where `C` is a constant can be thought of as involving a function `fC` which always returns `C`. `fC` can be treated as a binary operation that ignores its

parameters, and `fC(fC(v,E1),E2) = fC(C,E2) = C = fC(C,E1) = fC(fC(v,E2),E1)`. Thus, `v=C` is equivalent to `v fC=NULL`. Thus, arguments about reductions will cover the assignments of expressions that are constant for the duration of the loop.

4.2. OTHER SELF-COMMUTATIVE OPERATIONS

Other sources of self-commutative operations are set insertion (into a set other than `Set1`, ..., `Setm`) and deletion of a set element (from a set other than `Set1`, ..., `Setm`). For the insertion or deletion statement to be self-commutative, the elements inserted or deleted must be of the form `f(X1,...,Xm,v1,...,vt)` where `f` is a mathematical function that does not examine fields of `X1-Xm` modified by `Sm1` and the `vi` are either temporaries or variables not modified in `Sm1`. Note that any of the `vi` can be a set that is not modified by statements in `Sm1`. Arbitrary read-only queries can be evaluated against the set `vi` to compute `f`.

Another source of self-commutative statements are some abstract data type (ADT) method invocations. We assume that an ADT method only modifies the internal state of the ADT it was invoked on; it does not modify any values that can be accessed outside a method invocation on the same ADT. For an ADT method invocation to be self-commutative, all parameters to the ADT method being invoked must be of the form `f(X1,...,Xm,v1,...,vt)` where `f` is a mathematical function that does not examine fields of `X1-Xm` modified by `Sm1` and the `vi` are either temporaries or variables not modified in `Sm1`. Further, the optimizer must be aware that the method commutes with itself—that the instantiations of the method invocation in the unrolled program can be permuted.

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 a tree traversal can produce positional information. Thus, maintaining insertion order is unimportant in the first case, but crucial in the second.

4.3. SELF-COMMUTATIVE SEQUENCES OF OPERATIONS

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

```
(4.3) v op= f(X1, ... ,Xm,v1, ... ,vt);
```

in Sml. The vi must either be temporaries or variables not be modified by Sml. Then all of the definitions of v in Sml must be reductions and the reduction operators must be either identical or inverses. For example, 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 (4.3). If these conditions hold for all variables/delta-fields defined by reduction operations in Sml, we say that Sml is **self-commutative relative to reduction operations.**

If T is an ADT instance that invokes methods M_1, ..., M_p in Sml, then M_i must commute with M_j $\forall i, j\ 1 \le i, j \le p$. If two adjacent instantiations of Sml in an unrolled version of the program are swapped, then the M_i invocation in the second instantiation will move before the $M_1, \ldots, M_p$ 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

```
(4.4) D->Insert(X1->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 (4.4) to be self-commutative. If this condition holds, we say that Sml is **self-commutative relative to ADT operations.**

Statements in Sml 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 instantiation of Sml and delete it in the instantiation that followed; in which case, the set will not contain the element upon completion. If the instantiations were flipped, the final set would contain the element. If this condition

holds, we say that Sm1 is **self-commutative relative to set operations.**

In summary, suppose that Sm1 is straightline code and that each statement in Sm1 is self-commutative when viewed in isolation. Then Sm1 is self-commutative if it is **self-commutative relative to reduction operations, ADT operations, and set operations.**

If Sm1 is of the form

```
S1;
if (b(X1,...,Xm,v1,...,vt)) S2;
S3;
```

then Sm1 is self-commutative if the sequence of statements S1; S2; S3 is self-commutative, b is a mathematical function that does not examine fields of X1-Xm modified by Sm1, and the vi are either temporaries or variables not modified by Sm1. The rule for if-then-else and 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.

As mentioned previously, this is not a complete characterization of the class of self-commutative statements, but it is a safe one. We believe this characterization is sufficient for optimizing many programs.

4.4. PROOF THAT CHARACTERIZATION OF SELF-COMMUTATIVITY IS CORRECT

We wish to show that our characterization of a subclass of the set of self-commutative statements is correct. If Sm1 is **self-commutative relative to reduction operations, ADT operations,** and **set operations,** we wish to show that is is self-commutative. The "atomic" statement classes that we consider are: (1) reductions, (2) set insertions/deletions, and (3) ADT method invocations.

Since Set1-Setm are not modified (except perhaps by assignments to delta-fields) and the loop predicates have no side-effects and do not use or modify variables/delta-fields modified by Sm1, query (4.1) is equivalent to a pair of loops: the first of which produces a list of m-tuples of Set1-Setm pointers, the second which iterates through the list of m-tuples executing a statement syntactically similar to Sm1 that uses pointers contained in the m-tuples instead of the original iterator variables. This in turn can be thought of conceptually in an unrolled form—one instantiation of Sm1 for each element of the list (no loops), and the instantiations are in list order. If (4.1) were replaced by a join loop, only the order of elements in the conceptual list would change—a permutation

of (4.1)'s list would be produced.

To show that Sm1 is self-commutative, it suffices to show that an arbitrary pair of Sm1-like statements (instantiations of Sm1) can be permuted without changing the state that is reached after the second of them is executed. This suffices, because then for any list of instantiations, an arbitrary permutation of that list will produce the same program state (since an arbitrary permutation can be produced by repeatedly permuting adjacent pairs of statements). But then replacing the conceptual list in (4.1) with an arbitrary permutation of that list will yield the same program state. In this case, (4.1) and the corresponding join loop will reach the same state.

We assume that for any **if** statement

```
if (b(X1,...,Xm,v1,...,vt)) S
```

in Sm1, the statement S is an atomic statement. We can assume this without loss of generality, since the compound statement

```
if  (b(X1,...,Xm,v1,...,vt)) {Sa; Sb; };
```

can be rewritten equivalently as

```
if (b(X1,...,Xm,v1,...,vt)) Sa;
if (b(X1,...,Xm,v1,...,vt)) Sb;
```

(by hypothesis the predicate has no side-effects, the delta-fields are not examined by b, and the vi are not modified by Sm1—and hence are not modified by Sa or Sb). Similarly, the statement

```
if (b(X1,...,Xm,v1,...,vt))
   if (b1(X1,...,Xm,v1,...,vt))
      Sa;
```

can be rewritten equivalently as

```
if (b(X1,...,Xm,v1,...,vt) && b1(X1,...,Xm,v1,...,vt))
      Sa;
```

Repeating this process will eventually give us statements of the desired form. Thus, our proof will only consider programs made up of sequences of atomic statements and of **if** statements which contain a single atomic statement. In the proof, if S=S1;...;Sk is an instantiation of Sm1, we will say that Si is **from** Sm1 (Si is either atomic or an **if** statement containing a single atomic statement).

Any two reductions that define different variables/delta-fields x1 op1= E1 and x2 op2= E2 can be permuted if E1 does not use x2 and E2 does not use x1, because then there is no interaction between the two state-

ments. This condition is met because both reductions are **from** `Sm1`, and `Sm1` is **self-commutative relative to reduction operations.** Two reduction operations **from** `Sm1` on the same variable/delta-field, `x op1= E1` and `x op2= E2`, are equivalent to `x = (x op1 E1) op2 E2` if neither `E1` nor `E2` examine `x`. If, further, `op1` and `op2` are identical or inverses, the pair of reduction operations is also equivalent to `x = (x op2 E2) op1 E1`. Since `Sm1` is **self-commutative relative to reduction operations**, these conditions are met, and the reductions can be permuted. Thus, any two reductions **from** `Sm1` commute.

A reduction **from** `Sm1`, `x op1= E1`, will commute with an ADT method invocation **from** `Sm1`, `T->Mi(X1,...,Xm,v1,...,vt)`, since the invocation only affects `T`, `T` is not examined in `E1` (since `Sm1` is **self-commutative relative to reduction operations** and `T` is modified by `Sm1`), and `x` is not a parameter to `Mi` (since `Sm1` is **self-commutative relative to ADT operations**). Using a similar argument, a reduction **from** `Sm1` will permute with a set insertion/deletion **from** `Sm1`, a set operation with an ADT method invocation, a set operation with another set operation, and an ADT method invocation with another ADT method invocation.

Suppose `Sm1` is a sequence of atomic statements. Then, using induction, it is clear that an arbitrary pair of instantiations of `Sm1` such as `S11;...;Sk1` and `S12;...Sk2` can be permuted (`S11;...;Sk1;S12;...;Sk2` and `S12;S11;...;Sk1;S22;...;Sk2` reach the same state since `S12` will permute with `S11`,..., and `Sk1`, and so on). Thus, if `Sm1` could not contain **`if`** statements, we would be done. We need to show that an **`if`** statement containing a single atomic statement will commute with an atomic statement or another **`if`** containing an atomic statement.

Given

`S1 = if (b(X1,...,Xm,v1,...,vt)) S`

and `S2` (where `S2` is atomic), we know that `S` and `S2` commute. Thus, if `(b(X1,...,Xm,v1,...,vt))` evaluates to true, `S1` and `S2` can be commuted (since `b` is side-effect-free, and the variables and fields of iterator variables that it examines are not modified by `S2`). If `(b(X1,...,Xm,v1,...,vt))` evaluates to false, `S1` is equivalent to the NULL statement, and `S2` can commute with the NULL statement. Thus, `S1` and `S2` commute. A similar argument can be made if `S1` and `S2` are both **`if`** statements.

Thus, we have shown that an arbitrary pair of instantiations of `Sm1` can be permuted. Thus, `Sm1` is **self-commutative.**□

CHAPTER 5

IMPLEMENTATION OF THE TRANSFORMATIONS

This chapter will describe how the transformations of Chapter 3 can be implemented using a standard transformation-based optimizer. Section 5.1 describes the tree representation of set loops that is used by the optimizer. Section 5.2 will show how the transformations can be expressed as tree-rewrites. Section 5.3 describes how we used these techniques to build an optimizer for the AT&T Bell Labs *O++* [AGRA89] compiler. We used the resulting optimizing compiler to experimentally validate the ideas in this thesis. The experiments described in Section 5.4 show that this technique can significantly improve the performance of database programming languages.

5.1. 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 Chapter 3 in source-to-source form have straightforward analogues in AST-to-AST form. A typical AST representation of

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

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 (i.e. `divisioncnt++` and the `for E` loop). While this is a good representation from which to generate code, it is a poor one for database-style 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 (OptGen), 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 one operand for each statement directly contained in the set loop. Since we decided to use OptGen, 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 optimization 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 set loop:[1]

```
(5.2) for (X1 of Set1) suchthat (Pred11(X1)) {
          S11;
          ...
          for (Xi of Seti)  suchthat
          (Predi1(Xi) && Predi2(X1,...,Xi)){
             Si1;
             ...
             for (Xm of Setm)  suchthat
             (Predm1(Xm) && Predm2(X1,...,Xm)) {
                Sm1;
             }
             ...
             Si2;
          }
          ...
          S12;
       }
```

In (5.2), `Predi1(Xi)` $\forall i\ 1 \le i \le m$ is a predicate that only involves `Xi`, constants, and variables constant for the duration of (5.2). `Predi2(X1,...,Xi)` contains all clauses of the predicate belonging to the **for** `Xi` loop that are not contained in `Predi1(Xi)`. (If the original query does not have have predicates in this form, the system manipulates them into this form.) We represent the query (5.2) as the tree in Figure 5.1.

[1]A query may appear in `Si1` and/or `Si2`, so this pattern matches all set loops $\forall i\ 1 \le i \le m$. However, the optimizer whose implementation is described in this chapter rejects loops that have queries in `Si1` or `Si2`; allowing such queries would complicate the cost functions.

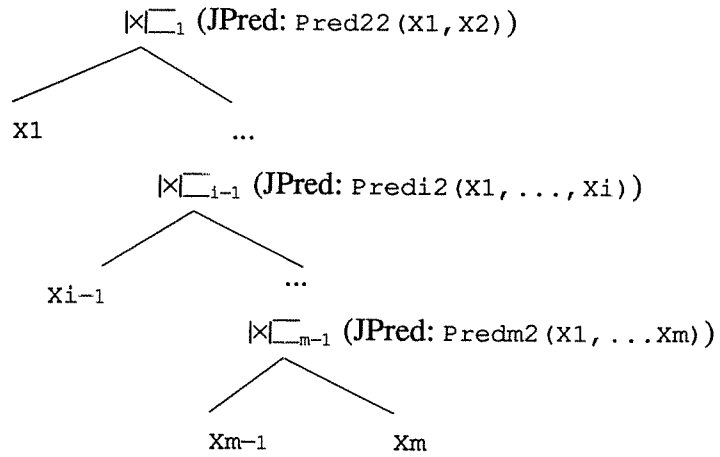

Figure 5.1: Tree Representation of (5.2)

The transformation process is mechanical. Each leaf node represents a set iteration. For example, the left-most node in Figure 5.1 corresponds to the `for X1` set loop in (5.2). We will identify a leaf node by the iterator variable of the corresponding set loop (e.g. the node corresponding to the `for X1` set loop in (5.2) is called the `X1` node). Figure 5.2 describes the contents of the `Xi` set node. The *First* field represents the code in the `for Xi` set loop that precedes the embedded `for Xi+1` set loop (e.g. in the `X1` node, since `S11` precedes the `for X2` loop, *First* has the value `S11`). 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*.

Var = `Xi`
First = `Si1`
Second = `Si2`
StmtsDesc = <*value described in text below* >
Flow = <*value described in text below*>
SetName = `Seti`
SelectPred = `Predi1(Xi)`

Figure 5.2: Contents of the `Xi` Set Node

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=Si1; Si2` would be self-commutative relative to a simple group-by loop over `Set1-Seti` with `S` as the inner statement[2], *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 `Set1-Seti` (or over temporary sets produced by applying selections and projections to `Set1-Seti`) containing just `S`, `Si1`, or `Si2` (or a version of

the statement S, Si1, or Si2 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 Xi+1 set loop or from the Xi+1 set loop to the **for** Xi loop[3]; *flow-in* means values flow from *First* or *Second* into the Xi+1 set loop, but no values flow from the Xi+1 set loop to *First* or *Second*[4]; *aggregate* means that the the Xi+1 loop calculates an aggregate (i.e. values flow from *First* to the Xi+1 loop and from the Xi+1 loop to *Second*, but in a constrained way)[5]; *innermost* means no set loop is contained in the Xi loop[6]. If output is produced in *First* or *Second* and in the Xi+1 loop, but there is no other flow of values from the Xi+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 $|\times|\sqsubset_{j-1}$ $\forall j$ $2 \le j \le n$ operator's left-hand child represents the **for** Xj-1 loop and all the statements contained in it except the **for** Xj loop. The right-hand child represents the complete **for** Xj set loop, except for those parts of the predicate that refer to the iterator variables Xi $\forall i$ $1 \le i \le j-1$ or to variables modified in the group-by loop (5.2). Those parts of the predicate (i.e. Predj2(X1,...,Xj)) are assigned to the *JPred* field of the $|\times|\sqsubset_{j-1}$ 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 Xi set node if any of the elements of Seti may be modified by the group-by loop. If so, the optimizations described below that involve projecting Seti to produce a temporary set and then using the temporary set instead of the original set must be suppressed. In this case, we need

---

[2]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 Chapter 4 where this is true. The first is a reduction operation on variable/delta-field v where v is used in the predicate of a surrounding loop. Not only are we not covering this case in this thesis, but 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, in most reasonable programs, the Xi loop will not delete object Xj $\forall j$ $1 \le j < i$, although the Xi loop may delete object Xi—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 a large class of plausible programs.

[3]See the Dept loop of query (5.8) for an example of a loop that is labeled *noflow* by the compiler.

[4]See Dept loop of query (3.19) for an example.

[5]See Professor loop of query (5.8) for an example.

[6]See Employee loop of query (5.1) for an example.

access to the original objects; access to projected sub-objects in a temporary set will not suffice. Since this extension is straightforward, we omit the details to simplify the exposition. We assume that none of the predicates have side-effects.[7] We also assume that the predicates do not use variables modified by program statements. This is not necessary—predicates involving such variables can be handled, but handling them requires complicating the representation and the exposition.

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 (For example, the value would be one for the `X1` node in (5.2). For the `X2` node, it would be the number of selected objects from `Set1` since the **`for`** `X2` loop will be executed once for each of them.) This makes it more difficult either to evaluate the cost functions (since trees must be evaluated top-down) or to apply the transformations (since information about how many times each node's set loop will be executed must be maintained).[8] However, this representation has two major advantages. First, it is easier to represent the transformations contained in this thesis 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`** `Xi` loop in query (5.2) $\forall i\ 1<i<n$. For example, in Figure 5.3, the left-deep representation of the **`for`** `X2` loop is enclosed in a square. Clearly, the loop does not correspond to any subtree of Figure 5.3.

[7]The implementation described in this chapter uses data flow analysis to ensure that optimization is not attempted for loops where predicates have side-effects.

[8]We have used both methods. In the implementation described in this chapter, we evaluate the cost functions top-down. Since OptGen only supports bottom-up cost evaluation, this forced us to produce a complex routine for cost functions. This routine violated the spirit of OptGen as it was non-modular; every time a new method was added to the system, this routine had to be modified. Future optimizer generator work should examine more flexible methods of cost evaluation than strict bottom-up.

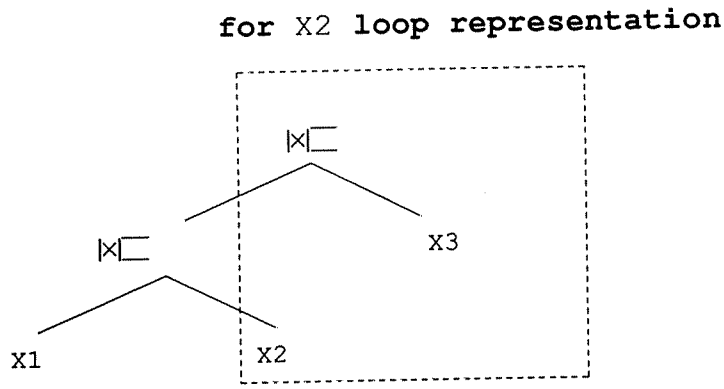

Figure 5.3: Left deep query representation of (5.2) when m=3

The *First* and *Second* fields are currently only used by one transformation during optimization; the fields *StmtsDesc* and *Flow* contain sufficient information for most transformations. 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.

5.2. 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

```
(5.3) for (X1 of Set1) suchthat (Pred11(X1))
        for (X2 of Set2) suchthat
        (Pred21(X2) && Pred22(X1,X2))
        {
           S21;
           for (X3 of Set3) suchthat
           (Pred31(X3) && Pred32(X1,X2,X3))
               S31;
           S22;
        }
```

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

```
(5.4) for (X1 of Set1; X2 of Set2) suchthat
      (Pred11(X1) && Pred21(X2) && Pred22(X1,X2))
      /*sort*/ by (X1->key)
      {
         S21;
         for (X3 of Set3) suchthat
         (Pred31(X3) && Pred32(X1,X2,X3))
            S31;
         S22;
      }
```

In (5.4), the join of `Set1` and `Set2` is produced and then sorted by the key of `Set1` (which may be its object identifiers). The observation that allowed query (5.3) to be transformed to (5.4) can also be represented using Figure 5.4. 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.

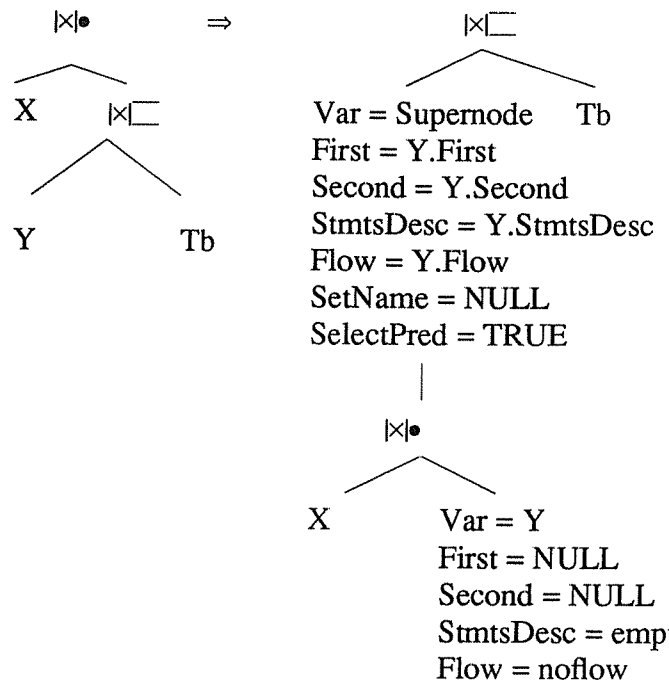

Figure 5.4: Supernoding

Since the operator on the left-hand side of Figure 5.4 is |×|•, the *First* and *Second* fields of the `X` node are NULL. This corresponds to the empty statements that precede and follow the **`for`** `X2` loop in query (5.3). We move the *First*, *Second*, *StmtsDesc*, and *Flow* fields of the `Y` node up to the Supernode, which represents the observation that the **`for`** `X1` and **`for`** `X2` 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:

```
(5.5) for (X1 of Set1) suchthat (Pred11(X1))
        for (X2 of Set2)  suchthat
        (Pred21(X2) && Pred22(X1,X2))
          for (X3 of Set3)  suchthat
          (Pred31(X3) && Pred32(X1,X2,X3)) {
            S31;
            for (X4 of Set4) suchthat
            (Pred41(X4) && Pred42(X1,X2,X3,X4))
              S41;
            S32;
          }
```

which can be represented using our tree notation as shown in Figure 5.5 (some unnecessary detail has been omitted).

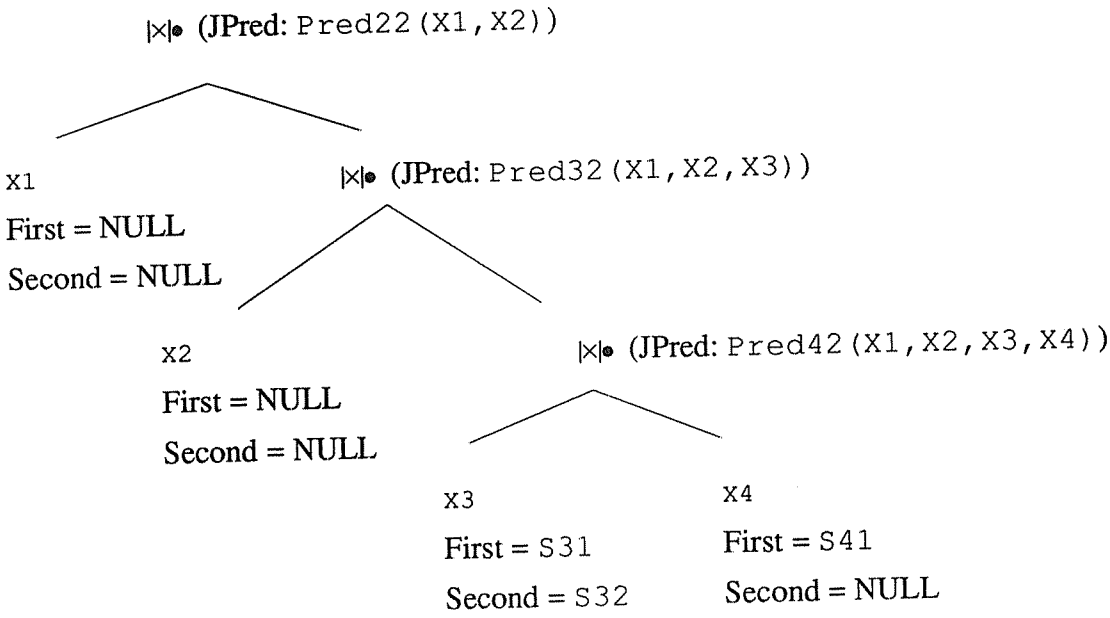

Figure 5.5: Tree Representation of query (5.5)

Applying the Supernoding transformation twice produces Figure 5.6 (the X1-X3 nodes have *First*=*Second*=NULL in Figure 5.6).

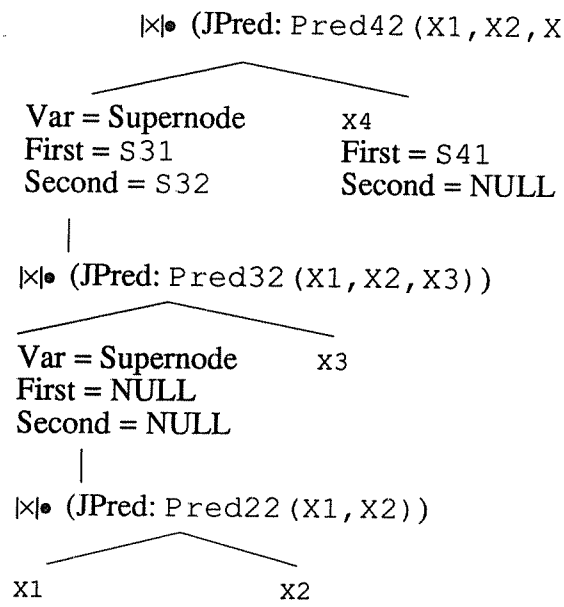

Figure 5.6: Tree Representation of query (5.5) after two applications of Supernoding transformation

Note that the tree representation in Figure 5.6 would closely resemble a standard tree representation for relational joins, the left-deep query tree [GRAE87], if the Supernodes were removed from the tree.

5.2.1. Simple group-by loops

Having introduced our basic terminology and the tree representation that is used for unoptimized queries, we next show how some of the transformations presented in Chapter 3 can be represented as tree rewrites. We will review the source-to-source form of the transformations and then examine the tree-to-tree form. The simplest loops that may be rewritten are of the form:

```
(5.6) for (X1 of Set1) suchthat (Pred11(X1))
        ...
         for (Xm of Setm) suchthat
         (Predm1(Xm) && Predm2(X1, ..., Xm))
             Sm1;
```

If `Sm1` is a self-commutative statement, then, by the definition of self-commutativity, query (5.6) is equivalent to:

```
(T1) for (X1 of Set1; ... ; Xm of Setm) suchthat
     (Pred11(X1) && ... &&
      Predm1(Xm) && Predm2(X1, ..., Xm))
          Sm1;
```

Even if `Sm1` is not self-commutative, if `Sm1` does not modify `Set1-Setm` or the variables used in the predicates, query (5.6) can be rewritten as a join followed by a sort:

```
(T2) Temp = {};
     for (X1 of Set1;...;Xm of Setm) suchthat
     (Pred11(X1) && ... &&
      Predm1(Xm) && Predm2(X1, ..., Xm))
        Insert <Needed(X1),...,Needed(Xm)> into Temp;
     Sort Temp on composite key (X1, X2, ..., Xm-1);
     for (T of Temp) //in sorted order
        Sml';
```

In transformation (T2), the `Temp` set loop contains a statement `Sml'` that looks like `Sml`, 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(Xi)` refers to the fields of `Seti` that either are used in statement `Sml` or are needed for the sort. It includes a unique identifier for `X1-Xm-1` for use in sorting; if the user has not supplied a primary key, `Needed(Xi)` includes the object identifier of `Xi` as the identifier. Chapter 3 explains the reason for the assymetric treatment of `Xm` in transformation (T2).

Since (T1) and (T2) are so similar, we will only show the tree transformation for (T2). (T2) can be represented as Figure 5.7. 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).

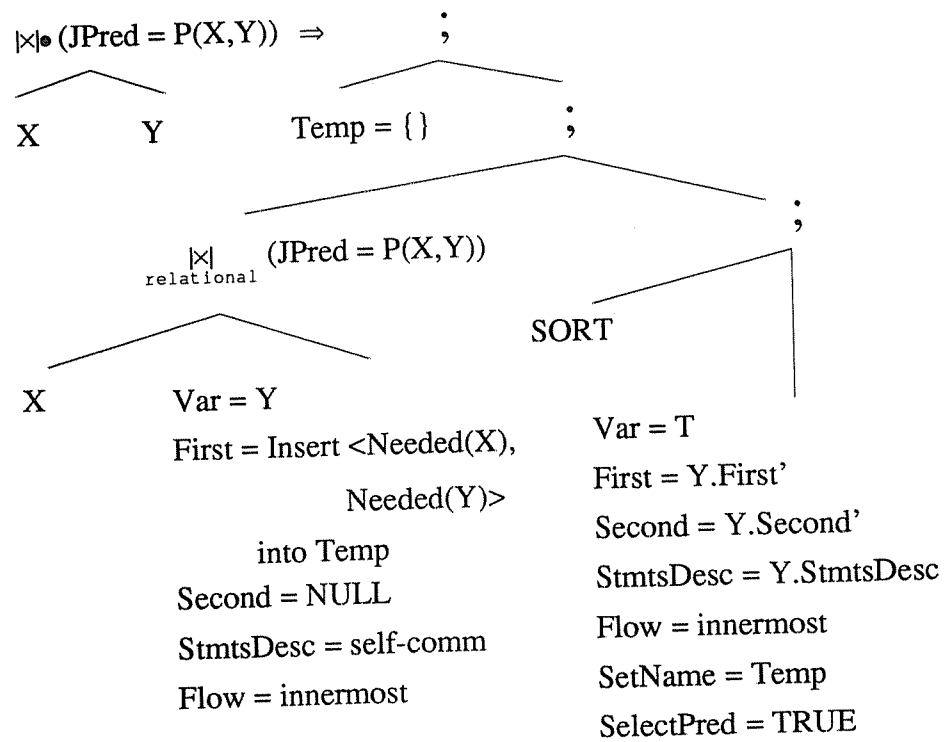

Figure 5.7: Transformation (T2)

The correspondence between (T2) and Figure 5.7 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 query (5.6). The $|\times|_{relational}$ subtree corresponds to the first set loop in (T2), the loop that inserts into `Temp`. The optimizer knows that the elements of the $X \; |\times|_{relational} \; Y$ join stream can be produced in any order. Remember that we have already made the tree on the left-hand side of Figure 5.7 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 $X \; |\times|_{relational} \; Y$ subquery. Since the standard relational transformations are well-known and have natural analogues here, we will not describe them in this thesis. The `SORT` node in Figure 5.7 corresponds to the sort in (T2). The *T* set node corresponds to the final loop in (T2)—in both, the system loops through the temporary set executing a modified version of the statement contained in the innermost loop.

5.2.2. General Group-by Loops

The group-by loops exemplified by query (5.6) 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

```
(5.7) for (X1 of Set1)  suchthat (Pred11(X1)) {
          S11;
          for (X2 of Set2)  suchthat
          (Pred21(X2) && Pred22(X1,X2))
              S21;
          S12;
      }
```

exemplifies the general case. In Chapter 3, four optimizations for general group-by loops were covered: (T3), (T4), (T5), and (T6). This section will only cover the tree-to-tree form of transformation (T4). Section 5.2.3 is an extended example of how the optimizer takes an original query plan and produces an optimized query plan. It applies transformations (T3) and (T5) along the way. Thus, this section and Section 5.2.3 will suffice to understand the optimizer representation for the four general group-by loop transformations.

Before we present the tree-to-tree form of transformation of (T4), we will review the source-to-source form of the transformation. Suppose the loops in query (5.7) do not interfere and no values flow from S21 to the outer loop (values may flow from S11 or S12 into the inner loop). Let v1, ... ,vn be the variables written by S11 and S12 that are used by expressions in the inner loop. Then query (5.7) can be rewritten as:

```
(T4) Temp = []; //empty sequence
     for (X1 of Set1) suchthat (Pred11(X1)) {
         S11;
         Append <Needed(X1), v1,...,vn> to Temp.
         S12;
     }
     for (T of Temp) //in insertion order
         for (X2 of Set2) suchthat
         (Pred21(X2) && Pred22(X1,X2)')
             S21';
```

S21' is S21 rewritten to use fields of Temp instead of vi $\forall i\ 1 \le i \le n$ and instead of fields of Set1. Pred22(X1,X2)' is Pred22(X1,X2) similarly rewritten. Needed(X1) contains the fields of X1 used in Pred22(X1,X2) and S21. 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, S21 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 rewrite shown in Figure 5.8. (*Tb'* means that the predicates and statements in tree *Tb* must be rewritten to use fields of Temp instead of vi $\forall i\ 1 \le i \le n$ and instead of fields of the outer set(s) included in the *X* subtree.)

Examples of where (T4) is profitable can be found in Chapter 3 and in Section 5.4 of this chapter.

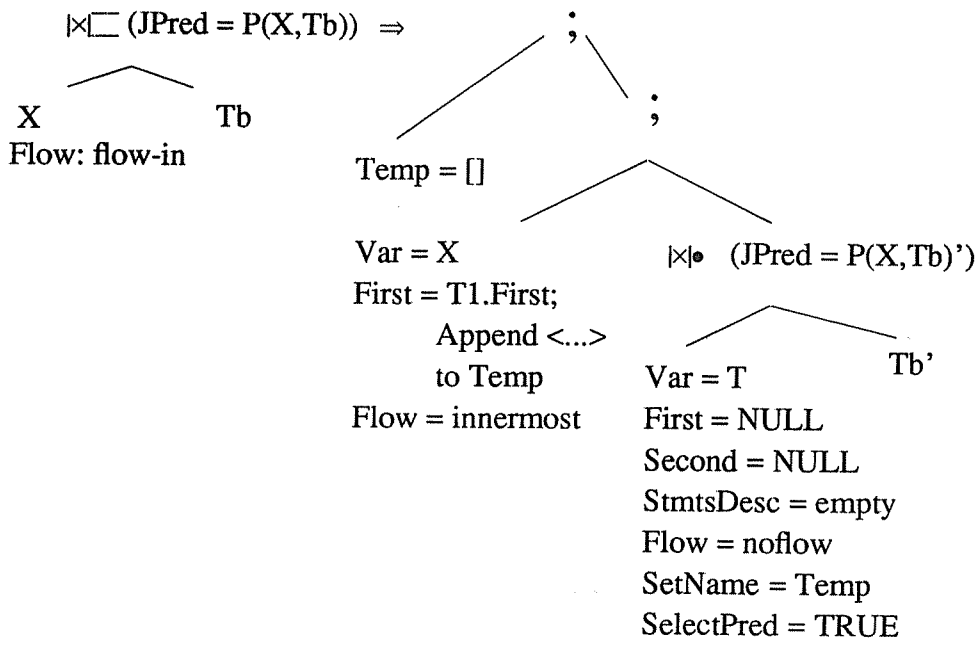

Figure 5.8: Transformation (T4)

5.2.3. Optimization Example

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

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

The optimizer first transforms query (5.8) into the tree representation that is shown in Figure 5.9 (the *SetName* and *SelectPred* fields are not included in our example trees).

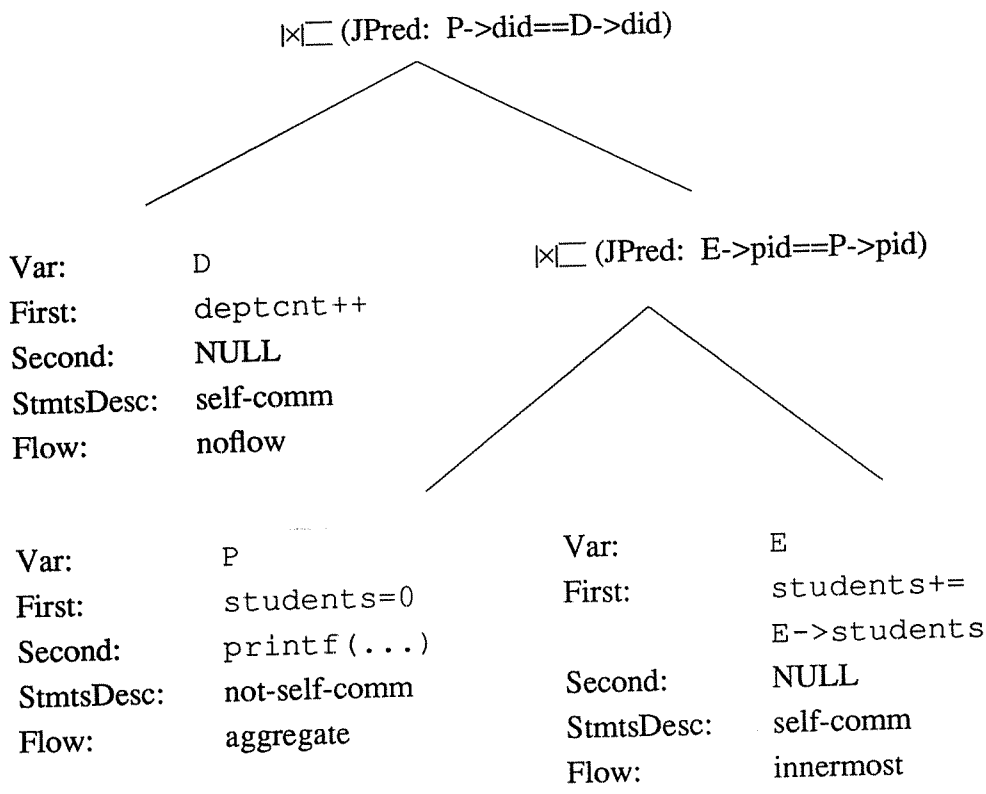

Figure 5.9: Tree Representation of query (5.8)

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 5.10:

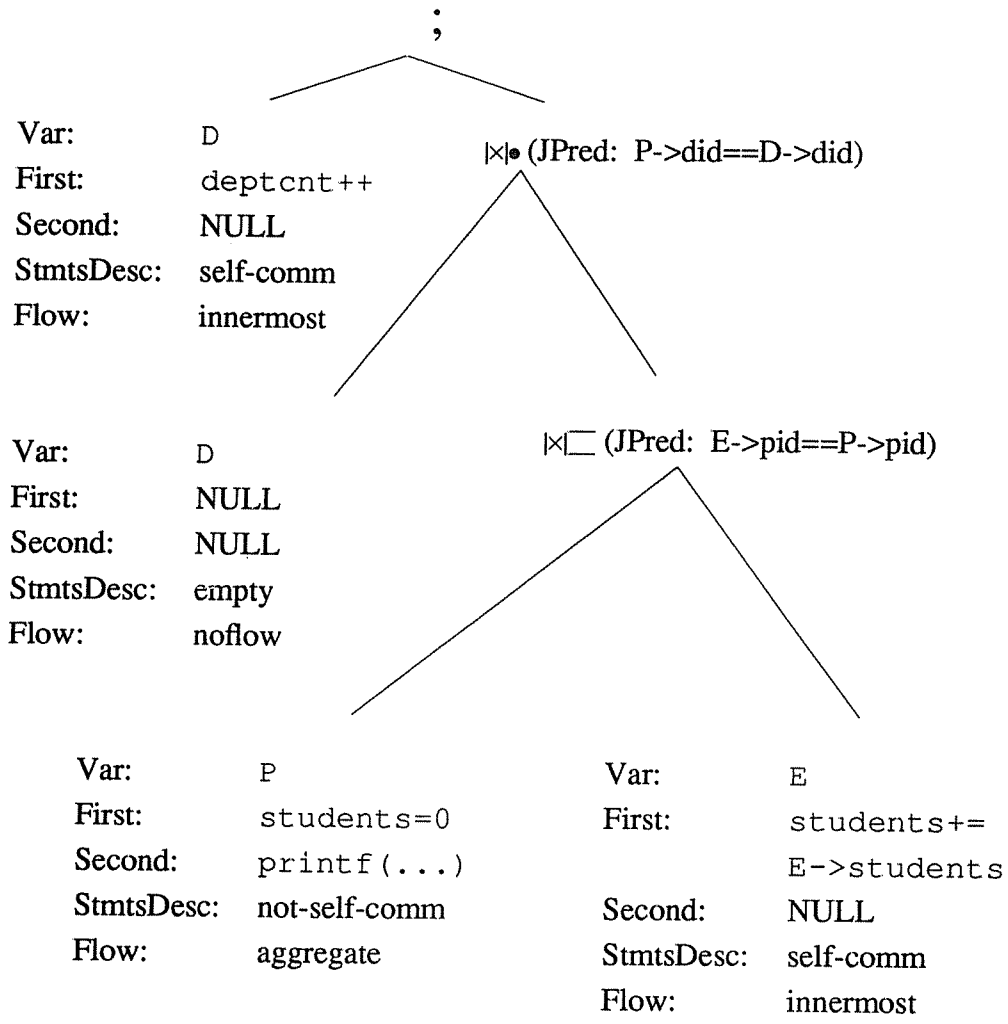

Figure 5.10: Tree Representation of query (5.9)

which corresponds to the following code:

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

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

The optimizer next considers the right-hand branch of the ; operator in Figure 5.10 (which corresponds to the second loop in query (5.9)). Since the D node's *StmtsDesc* field has the value *empty*, it applies the Supernoding

transformation to produce:

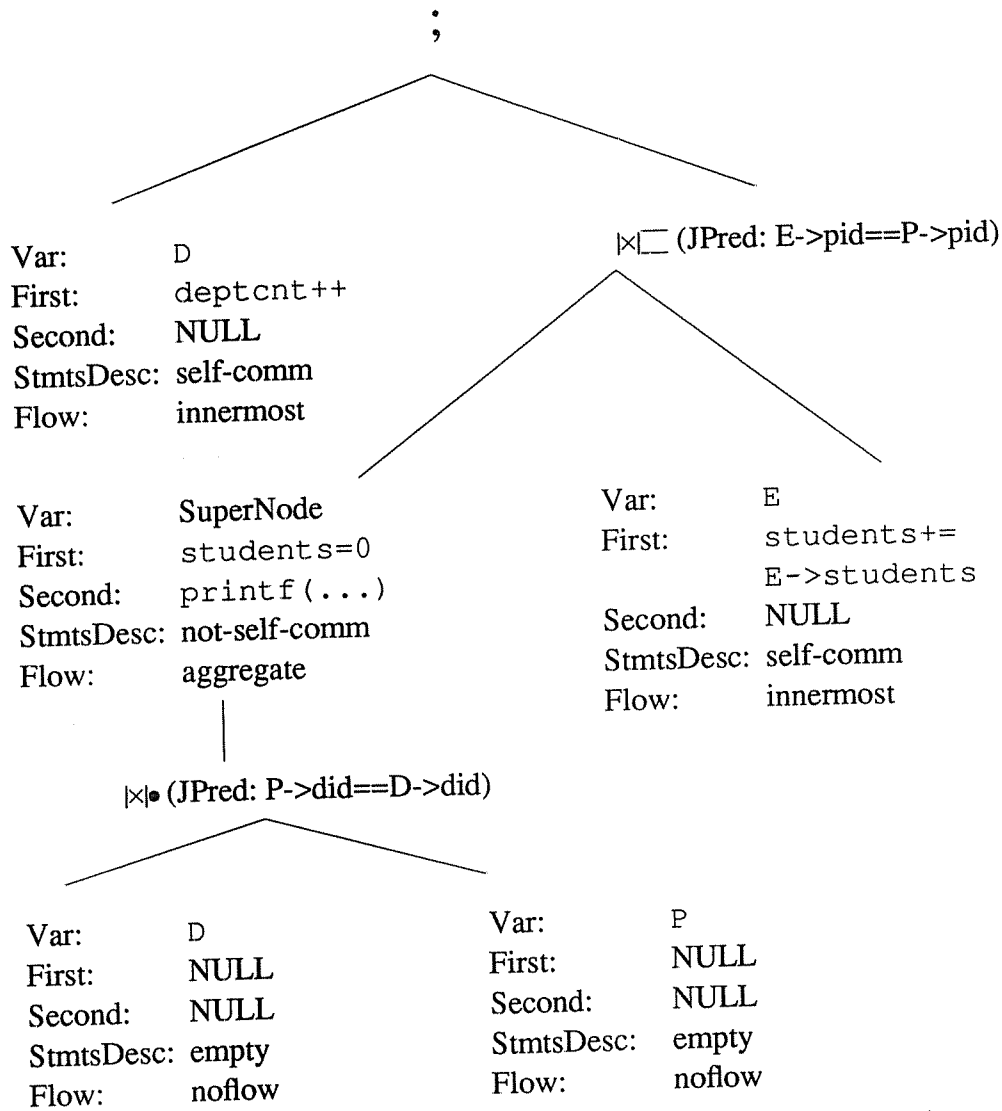

Figure 5.11: Alternative Tree representation of query (5.9)

Figure 5.11 also corresponds to query (5.9). 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 (5.9). After applying (T5) to the right branch of the ; operator in Figure 5.11, the following tree is produced:

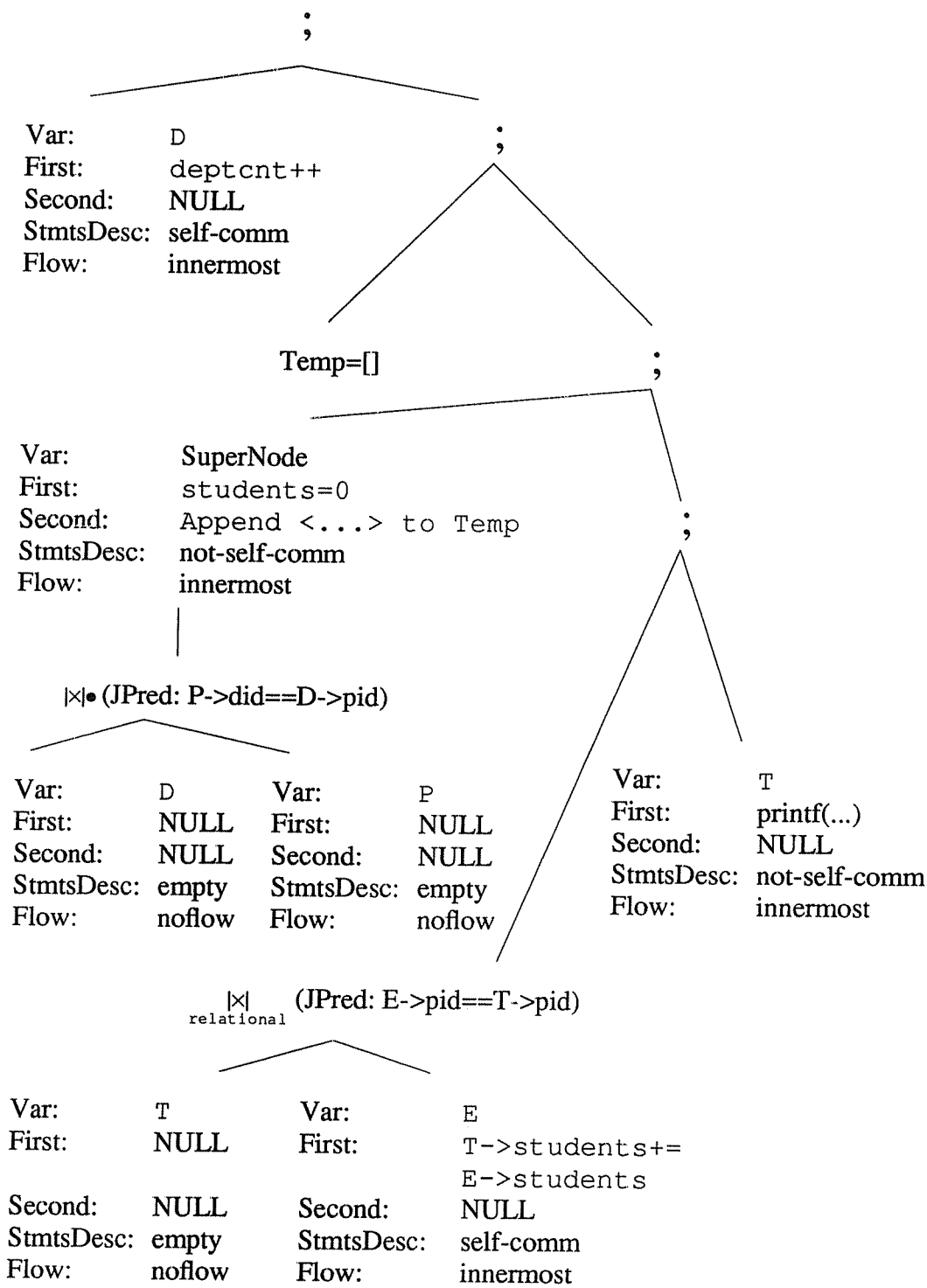

Figure 5.12: Tree representation of query (5.10)

Figure 5.12 corresponds to the code fragment:

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

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

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

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

The optimizer's final transformation of Figure 5.12 will be to choose a join method for the $|\times|_{\texttt{relational}}$ subtree.

5.3. SYSTEM ARCHITECTURE

We implemented these ideas in the *O++* compiler being developed at AT&T 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.

5.3.1. 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. To add the optimizer to the system, we added about twenty lines of code to one *O++* source code file and modified the `makefile` to link in the object code of the optimizer. The twenty lines of code make calls to two "black boxes": the optimizer (all new code) and the code generator for query trees (which did about half its work by making calls to existing routines for generating code from set loop ASTs). These two black boxes required about 10,000 lines of *C++* code.

The optimizer was built using the EXODUS Optimizer Generator. The transformations presented in Chapter 3 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 5.8 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. The system architecture of the implementation used here is illustrated by Figure 5.13. The right-hand side of the figure represents the compiler/optimizer part of the implementation.

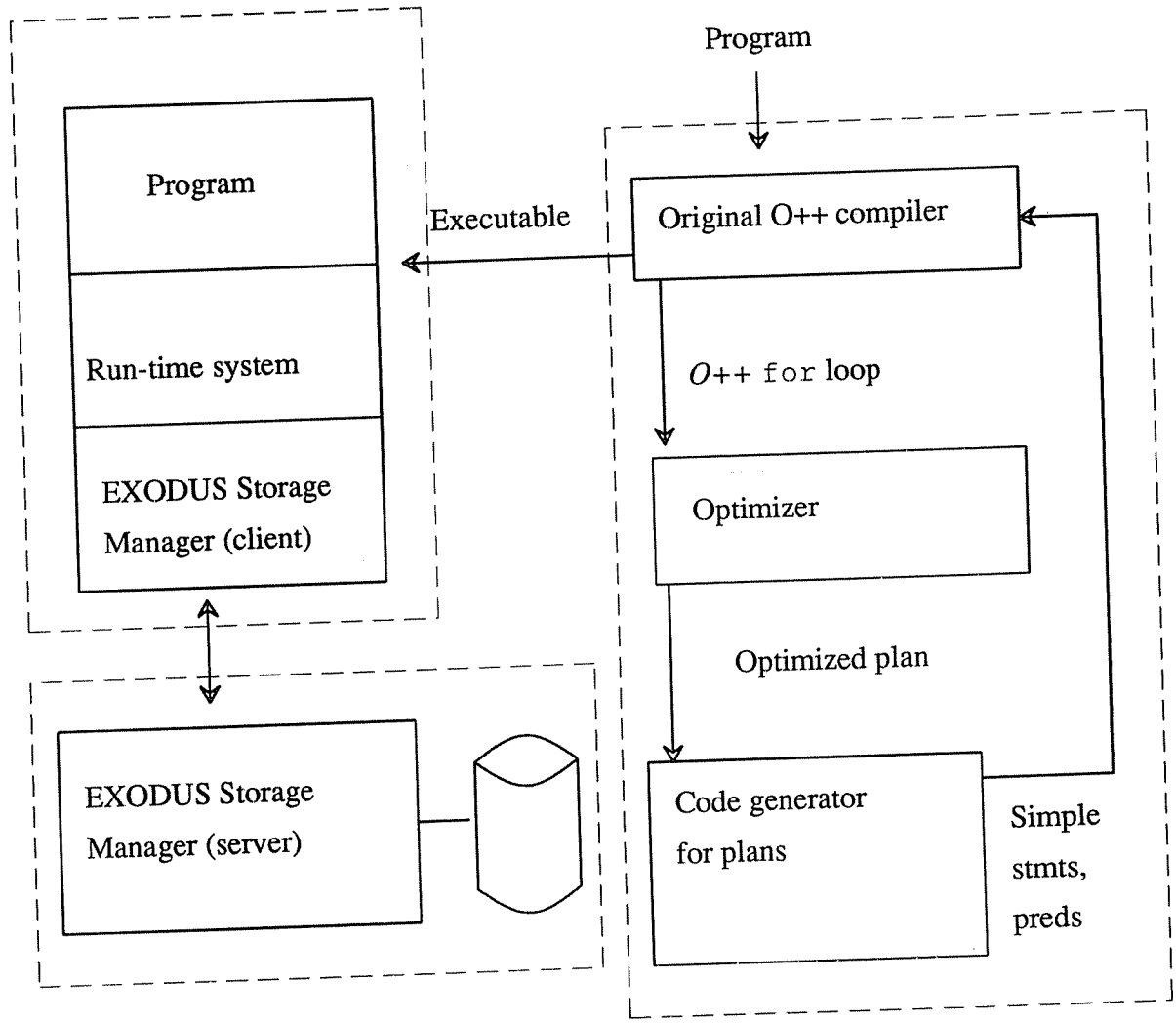

Figure 5.13: System Architecture of the Implementation

5.3.2. Run-time System

We used a different run-time system than the one that the original *O++* compiler generated code for.[9] That system copied all referenced objects into virtual 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, a different storage manager was needed for our purposes here.

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

We used Version 2.0.2 of the EXODUS Storage Manager [CARE86] to hold our test database.[10] 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 data. We didn't build the complete set of classes or interfaces provided by the *O++* storage manager; we built just enough to create data and to run read-only queries.

The original *O++* implementation did not provide indexes or any join algorithms other than tuple-at-a-time nested loops. We added indices to the system to allow more realistic experiments to be run. We plan to extend the system with more join methods in the future. Since unanalyzed group-by loops must be evaluated with a tuple-at-a-time nested loops join algorithm (without an index unless the set loop predicates are analyzed and found to be side-effect free), 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.

5.4. EXPERIMENTS

All experiments were performed using two identically configured SUN SPARCstation ELCs (approximately 20 mips). One was used as the client and the other was used as the server. The client and server machines were connected via a private Ethernet. Both machines had 24 megabytes of main memory. Data was stored at the server on two 69 megabyte raw disk partitions located on separate SUN0207 disk drives. One partition was used for the transaction log, and the other was used to store normal data. The client had a 2 megabyte (500 4K page) buffer pool; the server had a 600K (150 4K page) buffer pool.[11] The schema was as follows:

[10]Earlier experiments reported in [LIEU92] were run on Version 1.2 of the EXODUS Storage Manager. We moved to the newer version to have access to the B-trees provided by the new release.

[11]In [LIEU92], we used a 10 megabyte buffer pool on a DECstation 3100, and our buffer pool could hold approximately 70% of the database. To make our examples more realistic, we wanted to decrease the fraction of the database that fits in the buffer pool. Rather than increase the size of the database, we made the buffer pool smaller.

```
class Dept {
public:
   char Dname[30];
   int did;
   int profcount;
   // 188 bytes of other information
};

class Professor {
public:
   int pid;
   int did;
       // A Professor is in the Dept whose
       // did matches the Professor's did
   int salary;
   char Pname[30];
   // 184 bytes of other information
};

class Enroll {
public:
   char name[30];
   int pid;
       // A class (represented by an Enroll
       // object) is taught by the Professor
       // whose pid matches the pid of the
       // class
   int students;
   // 30 twelve byte pointers to students in class
   // 80 byte course description
};
```

Originally, we planned to average the response times of a number of repetitions of each experiment. However, after running a large number of tests on the machine, we found that the timings were almost identical from run-to-run, so the observed response times in the graphs presented here come from a single run. In the experiments, there were 400 `Dept`, 8000 `Professor`, and 24000 `Enroll` objects. Including objects and their overhead,[12] the `Dept` extent contained 25 pages, the `Professor` extent 500, and the `Enroll` extent 3000. There was an unclustered index on the `did` field of `Dept`, the `pid` field of `Professor`, and the `pid` field of `Enroll`. (The most common operations on the database are to find information about a particular `Dept` or `Professor`. Sometimes, when finding out information about the `Professor`, information about the classes the `Professor` teaches will also be desired. Thus, these indexes make the most common operations fast.) Each `Dept` had 20 `Professors`. Half the `Professors` taught two classes; the other half taught four. 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

[12]There is a twenty byte per object overhead for control information in the Storage Manager.

implemented optimizer.[13]

We consider the following query:

```
(5.11) for (D of Dept) suchthat (D->did>=lower) {
         deptcnt++;
         for (P of Professor) suchthat (P->did==D->did)
         {
            int students = 0;
            profcount++;
            for (E of Enroll) suchthat (E->pid==P->pid)
                 students += E->students;
            printf("%s %s %d",D->Dname,P->Pname,
                               students);
         }
       }
```

In order to simulate various selectivity factors on `Dept`, `lower` was varied in the experiments across the range 350 to 399 (`did` values in `Dept` ranged from zero to 399—thus, the various instantiations of the query selected between one and fifty `Depts`). Let sel_D be the number of selected `Dept` objects. A number of different execution plans for `(5.11)` are chosen by the optimizer, depending on the value of sel_D and on the type of optimization that is done. In the course of optimizing this query, the optimizer uses transformations (T2), (T4), (T5), and relational commutativity. It also uses indices in some of the implementations. Thus, this single query illustrates many of the techniques in this paper. 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 thesis can be implemented and can significantly improve performance.[14]

We optimized the query in three different modes: Transformations-only (where the optimizer uses all the transformations described in Chapter 3, but ignores the existence of indices—thus tuple-at-a-time nested loops is the only join algorithm used), Indices-only (where the optimizer considers using indices, but doesn't apply transformations), and Transformations-and-Indices (where the optimizer uses the full repertoire of techniques at its disposal). In practice, we do not expect to use these different modes; our aim was to demonstrate that both access-path-selection and the proposed Transformations contribute to producing better code. To do this, we compare the

[13]The names created by the compiler for the optimized code were simplified by hand.

[14]If more examples are desired, see [LIEU92]. [LIEU92] contains two other queries in the experimental sections, shows how they are optimized by an earlier version of the system (which did not have indices), and gives timing information for running the queries. One optimized implementation for each plan was shown. Producing the optimized versions described in this section required using all of the transformations used in the two examples in [LIEU92].

response time of the code generated in the three modes. The exact difference in performance of the generated code is unimportant; the point of this section is to demonstrate that combining access-path-selection and transformation produces better code than using either technique by itself. A secondary reason for using these three modes was to demonstrate that the optimizer considers a wide-variety of plans during optimization (since all of the plans in this section are considered in Transformations-and-Indices mode) and that it can produce rather sophisticated plans using our transformations.

The simplest plans were produced in Indices-only mode. When sel_D is between one and twenty-two, the optimizer chose to use an index on both `Dept` and `Enroll`. The following code is generated.

```
(5.12) for (D of Dept) suchthat (D->did>=lower) {
       //using the index on D->did
         deptcnt++;
         for (P of Professor) suchthat (P->did==D->did)
         {
            int students = 0;
            profcount++;
            for (E of Enroll) suchthat (E->pid==P->pid)
            //using the index on E->pid
                 students += E->students;
            printf("%s %s %d",D->Dname,P->Pname,
                                students);
         }
       }
```

When sel_D is twenty-three or larger, only the index on `Enroll` is used, so the following code is generated:

```
(5.13) for (D of Dept) suchthat (D->did>=lower) {
         deptcnt++;
         for (P of Professor) suchthat (P->did==D->did)
         {
            int students = 0;
            profcount++;
            for (E of Enroll) suchthat (E->pid==P->pid)
            //using the index on E->pid
                 students += E->students;
            printf("%s %s %d",D->Dname,P->Pname,
                                students);
         }
       }
```

The graph in Figure 5.14 shows the performance of queries (5.12) and (5.13), the implementations chosen in Indices-only mode. The response time for queries (labelled Total) was produced using the UNIX™ command `gettimeofday` before each query began and after it ended. User and system times were computed in the same way using the command `getrusage`. We summed those two times to calculate the CPU component of the

execution time. We subtracted the CPU time from the Total time to calculate the IO+network time. The curves for (5.12) and (5.13) have almost the same slope because the only difference between the plans is how they select elements of Dept. Since Dept is small and the selection is done only once, both plans do approximately the same amount of I/O and CPU work for a given sel_D value. The gaps in the curves indicate a change of plan from (5.12) to (5.13).

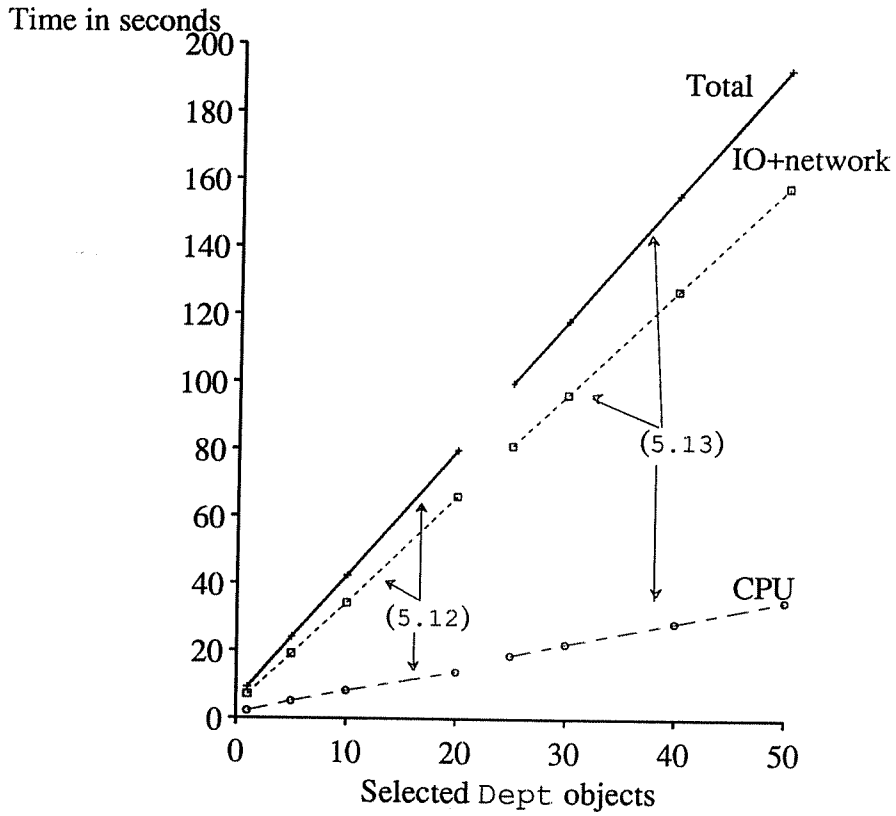

Figure 5.14: Indices-only mode

In Transformations-only mode, the optimizer applies (T4) to pull deptcnt++ out of the outer loop in query (5.11). Now there are two loops; the first is a loop over Dept to compute deptcnt++ and produce a selected, projected subset of Dept called $\text{Temp}_{\text{Dept}}$. The second is a loop over $\text{Temp}_{\text{Dept}}$, Professor, and Enroll. The optimizer then applies (T5) to the second loop to produce the following:

```
(5.14) Temp_Dept = [];
       for (D of Dept) suchthat (D->did>=lower)
       {
          deptcnt++;
          Insert <D->Dname,D->did> into Temp_Dept;
       }

       Temp_join = [];
       for (T of Temp_Dept)
          for (P of Professor) suchthat (P->did==T->did)
          {
             int students = 0;
             profcount++;
             Insert <T->Dname,T->did,[15] Unique(T),
                     P->Pname,P->pid,students>
                     into Temp_join;
          }

       for (T2 of Temp_join; E of Enroll)
       suchthat (E->pid==T2->pid)
             T2->students += E->students;

       for (T2 of Temp_join)
          printf("%s %s %d",T2->Dname,T2->Pname,
                             T2->students);
```

(`Unique(T)` is an integer that represents the position of a `Temp` object in the `Temp` sequence; since the position of an object in a sequence does not change unless the sequence is sorted, this can be used as a unique identifier for a `Temp` object. An integer requires less space than an OID.) The optimizer then applies relational commutativity to the third loop in query (`5.14`). $Temp_{join}$ becomes the inner set, and the following is produced:

[15]The optimizer does not currently project temporary sets. If we did, `T->did` would not be included. This will be a future enhancement to the system that will further improve performance.

```
(5.15) Temp_Dept = [];
       for (D of Dept) suchthat (D->did>=lower)
       {
          deptcnt++;
          Insert <D->Dname,D->did> into Temp_Dept;
       }

       Temp_join = [];
       for (T of Temp_Dept)
          for (P of Professor) suchthat (P->did==T->did)
          {
             int students = 0;
             profcount++;
             Insert <T->Dname,T->did,Unique(T),
                     P->Pname,P->pid,students>
                     into Temp_join;
          }

       for (E of Enroll)
          for (T2 of Temp_join)
          suchthat (E->pid==T2->pid)
             T2->students += E->students;

       for (T2 of Temp_join)
          printf("%s %s %d",T2->Dname,T2->Pname,
                             T2->students);
```

This is the plan chosen by the optimizer in Transformations-only mode when sel_D is one. When sel_D is two or more, the optimizer applies (T2)[16] to the second loop in (5.15) to make $Temp_{Dept}$ the inner set. The following is produced:

[16]Actually, it applies a variant of (T2) which takes a simple group-by loop that creates a sequence and converts it into a join loop that creates a set and then sorts the set to produce a legal sequence. Unlike (T2), the variant does not need to iterate through the sequence produced.

```
(5.16) Temp_Dept = [];
       for (D of Dept) suchthat (D->did>=lower)
       {
          deptcnt++;
          Insert <D->Dname,D->did> into Temp_Dept;
       }

       Temp_join = [];
       for (P of Professor)
          for (T of Temp_Dept) suchthat (P->did==T->did)
          {
             int students = 0;
             profcount++;
             Insert <T->Dname,T->did,Unique(T),
                     P->Pname,P->pid,students>
                     into Temp_join;
          }

       Sort Temp_join by Unique(T);

       for (E of Enroll)
          for (T2 of Temp_join)
          suchthat (E->pid==T2->pid)
             T2->students += E->students;

       for (T2 of Temp_join)
          printf("%s %s %d",T2->Dname,T2->Pname,
                             T2->students);
```

Figure 5.15 shows the performance of queries (5.15) and (5.16), the implementations chosen in Transformation-only mode. These implementations read all the pages of Dept, Professor, and Enroll just once. In fact, they do exactly the same amount of I/O across the sel_D range of one to fifty because $Temp_{join}$ fits in main memory; they are CPU bound. The response time increases as sel_D increases almost entirely because of the increased amount of CPU work in the third loop—since |Enroll|=24000 and the Enroll|×|$Temp_{join}$ loop iterates through $Temp_{join}$ once for each Enroll object, the Enroll|×|$Temp_{join}$ loop requires a great deal of CPU time. The Indices-only implementations do many fewer I/Os for low sel_D values than do the Transformation-only implementations. However, the Indices-only I/O cost grows faster than the Transformations-only CPU cost, which is why Indices-only implementations become more expensive than Transformation-only implementations.

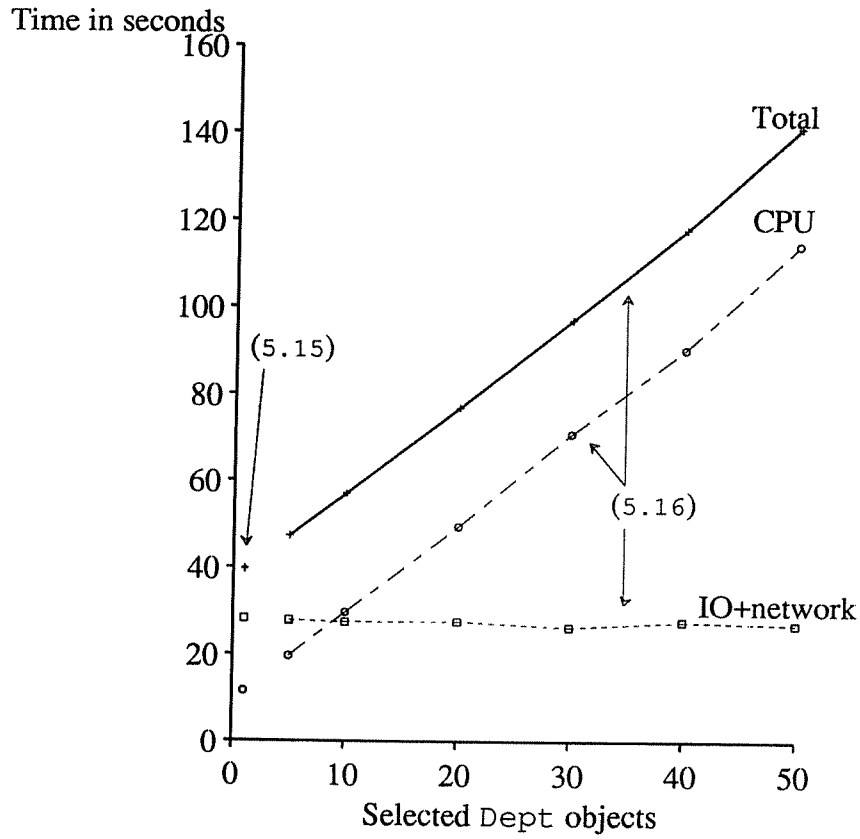

Figure 5.15: Transformations-only mode

When the optimizer is in Transformations-and-Indices mode and sel_D is one, query (5.12) is produced. When sel_D is between two and twenty-two, the optimizer takes query (5.14) and applies (T2) to the second loop. It then decides to use an index on Dept for the first loop and to implement the third loop by index-nested-loops with Enroll as the inner set. The following code is produced:

```
(5.17) Temp_Dept = [];
       for (D of Dept) suchthat (D->did>=lower)
       //using the index on D->did
       {
          deptcnt++;
          Insert <D->Dname,D->did> into Temp_Dept;
       }

       Temp_join = [];
       for (P of Professor)
          for (T of Temp_Dept) suchthat (P->did==T->did)
          {
             int students = 0;
             profcount++;
             Insert <T->Dname,T->did,Unique(T),
                     P->Pname,P->pid,students>
                     into Temp_join;
          }

       Sort Temp_join by Unique(T);

       for (T2 of Temp_join)
          for (E of Enroll) suchthat (E->pid==T2->pid)
          //using the index on E->pid
             T2->students += E->students;

       for (T2 of Temp_join)
          printf("%s %s %d",T2->Dname,T2->Pname,
                             T2->students);
```

When sel_D is twenty-three or larger, the system chooses the same plan as query (5.17) except that the index scan of Dept is replaced with a scan of the whole Dept extent. The following is produced:

```
(5.18) Temp_Dept = [];
       for (D of Dept) suchthat (D->did>=lower)
       {
          deptcnt++;
          Insert <D->Dname,D->did> into Temp_Dept;
       }

       Temp_join = [];
       for (P of Professor)
          for (T of Temp_Dept) suchthat (P->did==T->did)
          {
             int students = 0;
             profcount++;
             Insert <T->Dname,T->did,Unique(T),
                     P->Pname,P->pid,students>
                     into Temp_join;
          }

       Sort Temp_join by Unique(T);

       for (T2 of Temp_join)
          for (E of Enroll) suchthat (E->pid==T2->pid)
          //using the index on E->pid
             T2->students += E->students;

       for (T2 of Temp_join)
          printf("%s %s %d",T2->Dname,T2->Pname,
                             T2->students);
```

Figure 5.16 shows the performance of queries (5.17) and (5.18), the implementations chosen in Transformations-and-Indices mode.

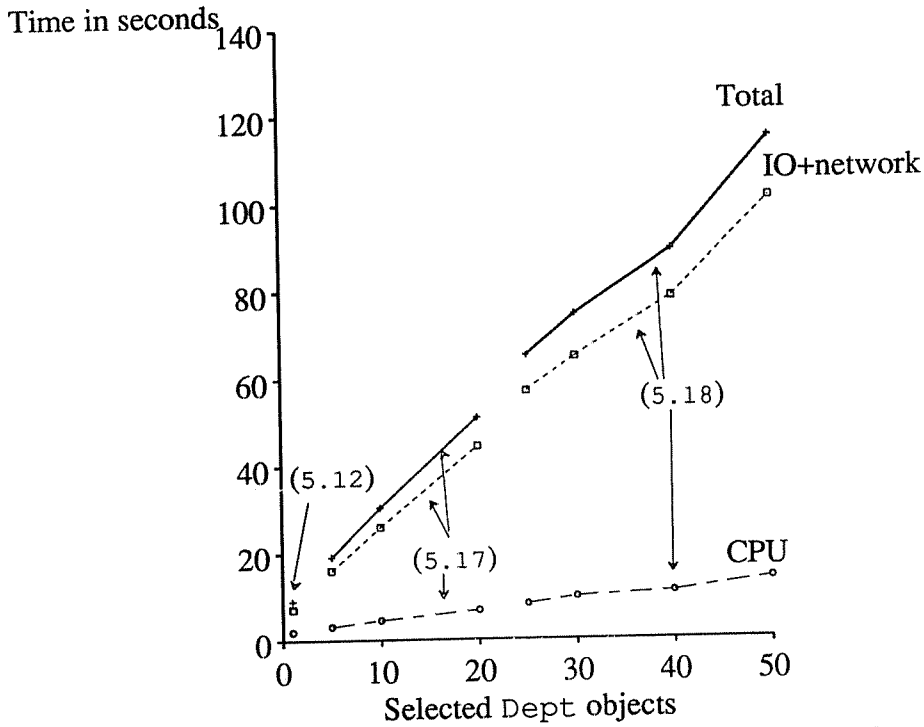

Figure 5.16: Transformations-and-Indices mode

Figure 5.17 compares the performance of all the plans in this section. Transformations-and-Indices produces faster code than Indices-only code because its plans read `Professor` once, while Indices-only plans reread 66 pages of `Professor` once for each selected `Dept`.[17] Both Transformations-and-Indices and Indices-only would have benefited equally if the index on `Enroll->pid` had been a clustered index since both probe the index the same number of times. Also, had the index been clustered, the response time for the Indices-only plans might have remained below that for Transformations-only plans across the sel_D range studied here. However, absolute performance is not the point of this section. Rather, we demonstrated that combining the use of transformations with access-path-selection can produce better code than using either technique by itself. We also demonstrated that our prototype optimizer can successfully produce rather sophisticated plans using the transformations.

[17]The system allocates 50 pages (the current maximum allocation of pages to an inner set of an index-nested-loops join) to the `Enroll` iterator and five pages to the `Dept` iterator. The `Professor` loop is allocated a 435 page buffer group with a MRU replacement policy—thus 434 pages of `Professor` are read once; the other 66 pages must be reread sel_D times.

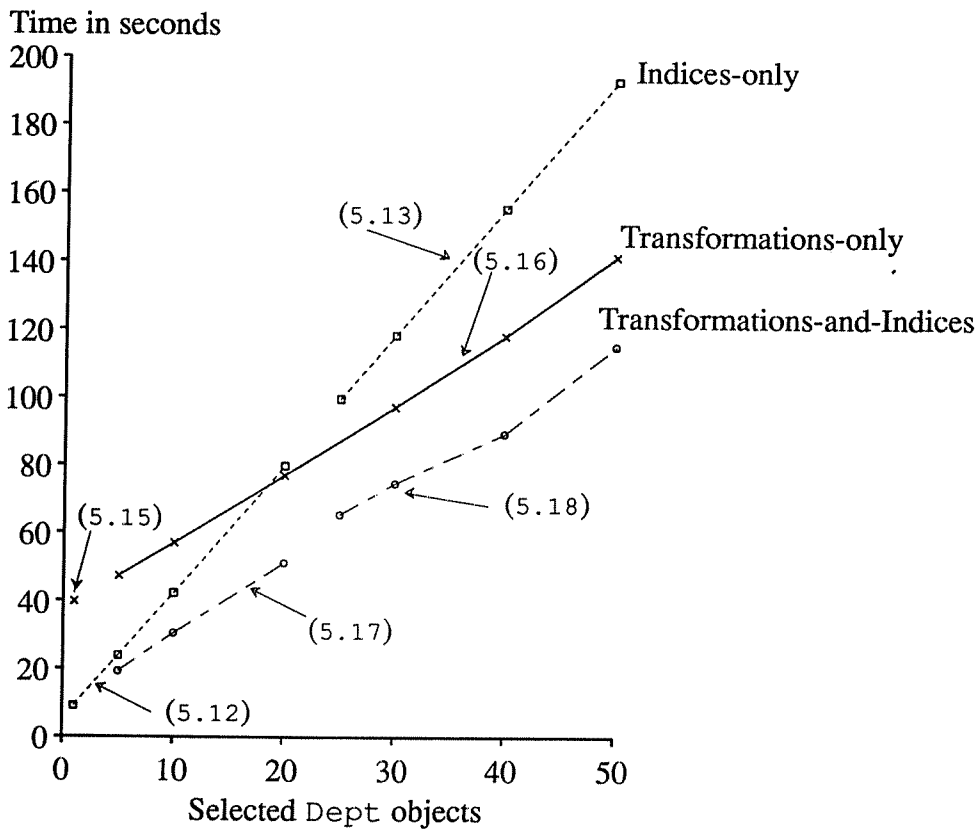

Figure 5.17: Response time for optimized versions of query (5.11)

To gain perspective on Figure 5.17, consider the performance of the code produced when no optimization (other than smart allocation of buffer pool pages to iterators) is attempted. In this case, query (5.11) allocates 480 pages to the Enroll iterator and five pages each[18] to the Professor and Dept iterators. Thus, essentially all of Professor must be read once for each selected Dept, and most of Enroll must be read once for each Professor that joins with a Dept. As can be seen in Figure 5.18, the performance of Unoptimized code is several orders of magnitude worse than the performance of the code produced in any of the optimization modes.

[18]Five pages is the minimum "safe" allocation for a buffer group in the EXODUS Storage Manager [ZWILL92].

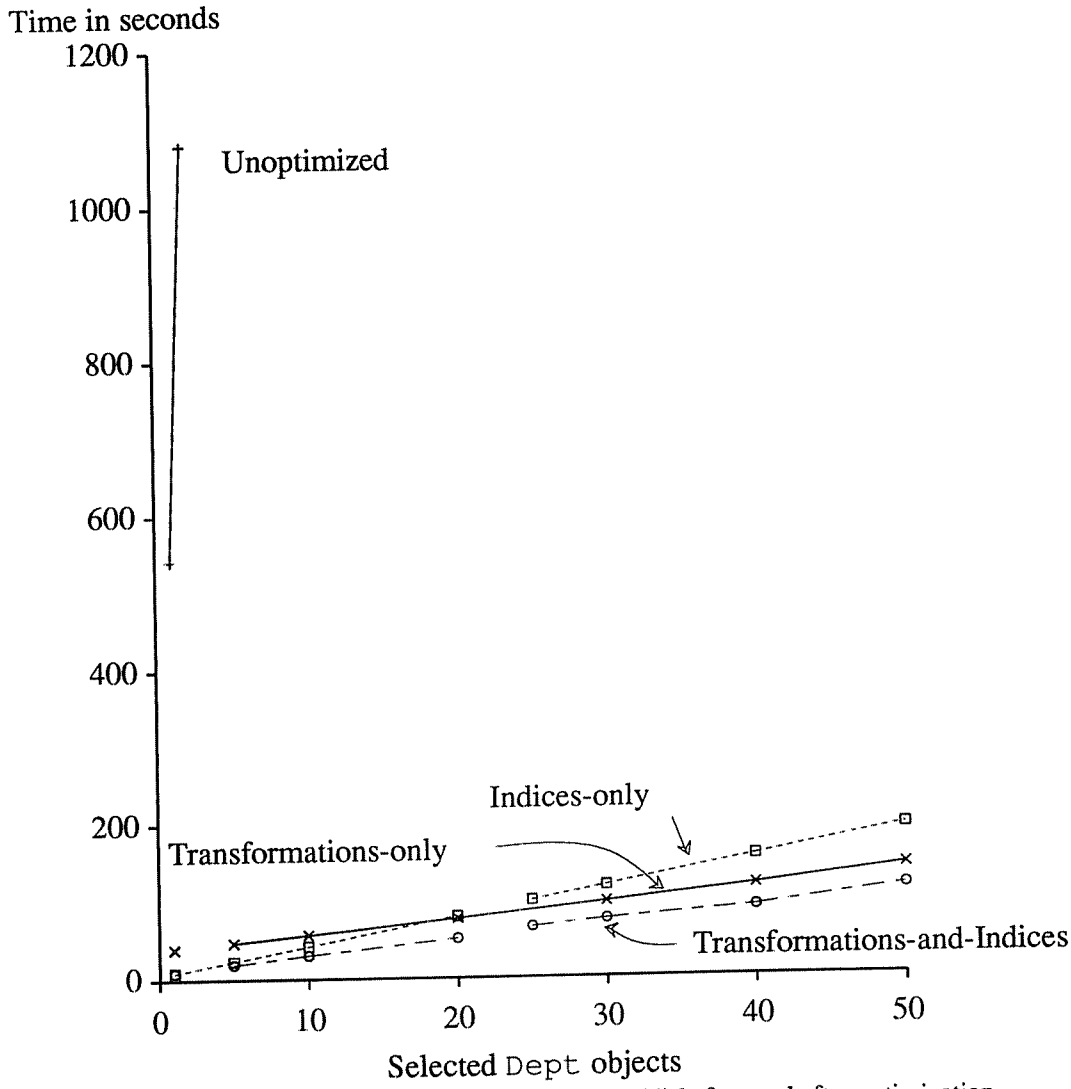

Figure 5.18: Response time for query (5.11) before and after optimization

Clearly, Figure 5.18 demonstrates that some optimization is required to produce acceptable performance. All optimization require some analysis. However, Indices-only optimization requires fairly simple analysis—it must be verified that the predicates associated with a set iteration statement are side-effect free and that none of the variables used in the predicates are modified in the loop—so Indices-only optimization is simpler to implement than the transformations we have developed. Thus, the decision of several commercial object database vendors [DEUX91, LAMB91, LECL89] to supply indices before supplying transformations is a reasonable one. However, as demonstrated above, to maximize performance, both access-path-selection and transformations are ultimately needed.

5.5. SUMMARY

This chapter described how standard transformation-based technology can be used to optimize group-by loops. First, our representation of a generic group-by loop as a tree was described. Then, we discussed how Chapter 3's transformations can be represented as tree-rewrites. Next, the steps the optimizer goes through (along a single transformation path) to transform a group-by loop involving three sets into a better plan were traced. Both the optimizer's tree representation of the plan and the corresponding *O++* [AGRA89] code were described for each step.

We then described an implementation of these transformations for the *O++* compiler. Only minor changes to the original compiler were required—the new code was localized in two "black boxes": an optimizer built using the EXODUS Optimizer Generator [GRAE87] and a code generator for plans. A brief description of a new run-time system built on top of the EXODUS Storage Manager [CARE86] was also given.

The resulting optimizing compiler was used to empirically demonstrate that our transformations can significantly improve the performance of a DBPL. The experiments also demonstrate that both access-path-selection and transformations are needed to maximize performance.

CHAPTER 6

PARALLELIZING LOOPS IN DATABASE PROGRAMMING LANGUAGES

Without program analysis, joins that are expressed as nested set loops must be evaluated using a tuple-at-a-time nested-loops join algorithm [SELI79] because otherwise program semantics may be violated. In addition, since loops have a sequential semantics, they must be executed at a centralized site. This chapter demonstrates how the analysis and transformations of Chapter 3 can be used to produce better parallel code than using a straightforward parallelization of a program's nested iterators.

This chapter is organized as follows. Section 6.1 contains boundary cases for the amount of parallelism that can be extracted from a program. Section 6.2 uses the concept of self-commutativity and some analysis of the flow of values through a program to parallelize both simple group-by loops and more complicated group-by loops. Section 6.3 presents and analyzes algorithms for computing joins involving set-valued attributes.

6.1. PARALLELISM BOUNDARY CONDITIONS

It is well known that relational joins can be executed in a highly-parallel manner [DEWI92a]. Many join loops can be parallelized to the same extent by modifying parallel join algorithms so that they execute program statements rather than produce result tuples. As an example, consider parallelizing query (6.1) using a modified **Hybrid-hash** algorithm [GERB86, SCHN89, DEWI92a]:

```
(6.1) for (D of Dept; P of Professor) suchthat (D->did==P->did) {
            Insert <D->name,P->name,P->sal> into PayInfo;
            D->budget += P->sal;
            profcount++;
      }
```

To simplify the description of the parallelization of query (6.1), assume that the `Dept` and `Professor` objects are declustered [RIES78, LIVN87, GHAN90] across the same n nodes, that these n nodes will be used to execute the join, and that all of the `Dept` objects will fit in the aggregate memory of the n processors. Then, query (6.1) can be executed at each node$_i$ $\forall i$ $1 \le i \le n$ as follows:

(0) Create a local variable `profcount`$_i$ and set it to zero.

(1) Repartition `Dept` by hashing on the `did` field of each `Dept` object to determine which node the object (tagged with its oid) should be sent to. As the objects arrive at node$_i$, insert them into the local `Dept` hash table, `Table`$_i$.

(2) Repartition the `Professor` set using the same hash function. As `Professor` objects arrive at node$_i$, probe `Table`$_i$ for matches. For each match, do the following (where `D` is a pointer to the matching `Dept` object in the hash table, and `P` is a pointer to the probing `Professor` object):

(a) Produce an object `<D->name,P->name,P->sal>` and write it to the local partition of `PayInfo`.[1]

(b) `D->budget += P->sal` (modify the `Dept` object in the hash table).

(c) Increment `profcount`$_i$.

(3) Repartition the `Dept` hash table, `Table`$_i$, using the oid of the `Dept` object to determine which node the object came from.[2] Send the object to that node. Write all arriving `Dept` objects to disk. Send `<DONE,profcount`$_i$`>` message to the application program.

The application program then sums the `profcount`$_i$'s values that it receives from each of the nodes. Since the application program requires notification from each node participating in the join when it completes, sending the `profcount`$_i$'s does not require any extra messages. The steps necessary to execute query (`6.1`) in parallel are shown in Figure 6.1 (where steps (0)-(3) shown at Node$_1$-Node$_n$ are those shown above).

---

[1]For simplicity, we assume that this will not violate the declustering strategy chosen for `PayInfo` and that either `PayInfo` is a multi-set or that each of the tuples is distinct (otherwise, duplicates must be eliminated).

[2]This assumes that the oid specifies a node. If the oid doesn't specify a node, we must tag each `Dept` object with a node identifier as well as an oid in step (1).

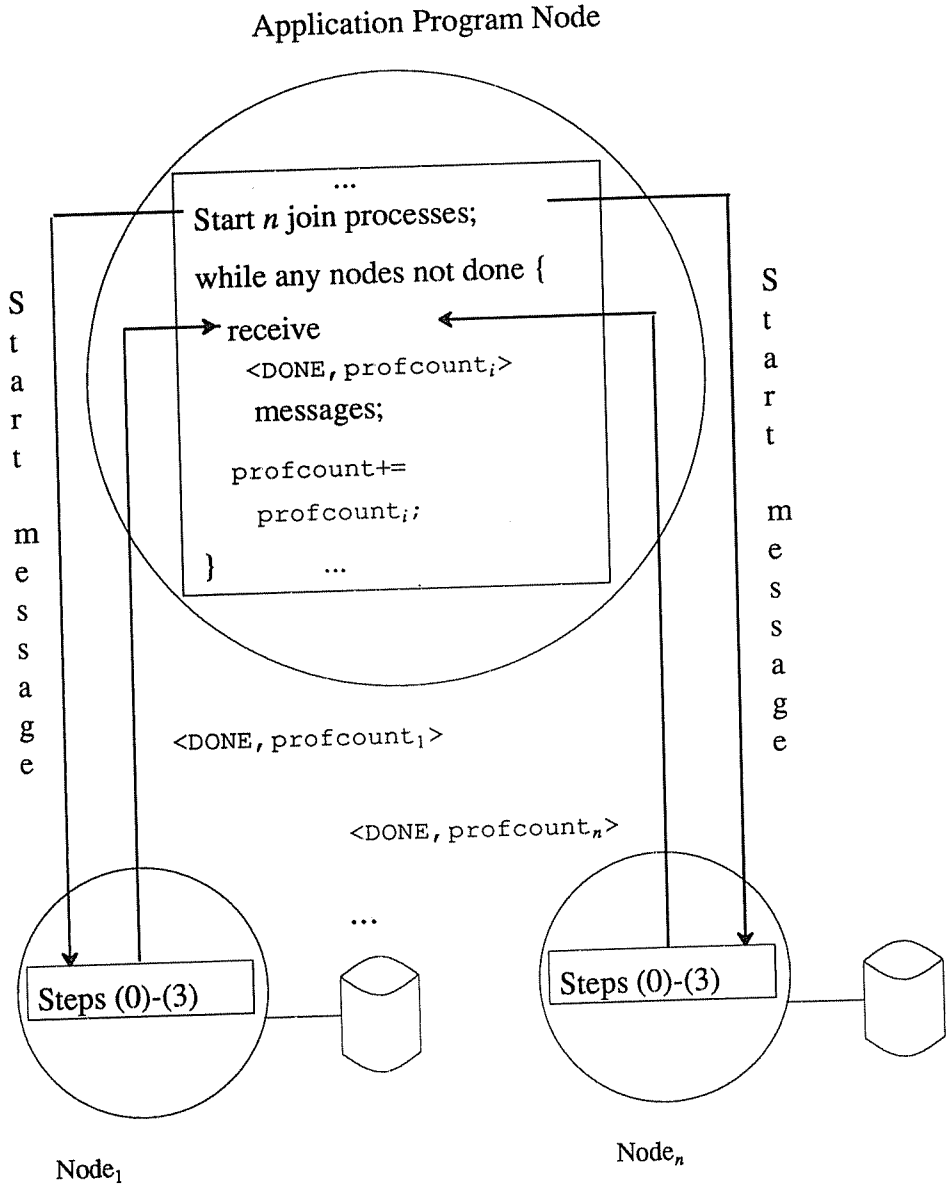

Figure 6.1: Parallelization of query (6.1)

Query (6.1) can be parallelized to the same extent as a relational join because the statements contained in the join loop fit a certain pattern. When the only operations in the statement list of a join loop are reductions and insertions into (or deletions from) sets other than those being iterated over,[3] result tuples do not need to be sent to the application program. A set insertion can communicate directly with the node that is to store the new set element; a

[3]As mentioned in Chapter 3, insertion into a set being iterated over specifies a fixpoint query. Thus, other techniques are more appropriate. The description of query (6.2) will demonstrate that deleting from a set being iterated through may not be parallelizable.

set deletion can communicate directly with the node containing the element that is to be destroyed. (Depending on the partitioning strategy and the description of the element to be inserted/deleted, a set operation may have to communicate with more than one node. Multi-set insertion can always be performed by communicating with a single node however.) A reduction operation on (or an assignment of an expression that is constant for the duration of the loop to) an element of a set not being iterated over can also be handled by communicating directly with the node containing the object being modified.[4] Similarly, a reduction operation on (or an assignment of an expression that is constant for the duration of the loop to) a field of an object from one of the sets participating in the join can be executed at the node performing the join with that object. (Actually, if the object participating in the join is replicated, the operation must be executed on the node from which the object came.[5]) Other reductions can make local copies of transient program variables defined by the reduction operation and operate on them. After the join is complete, the results of these reduction sub-operations can be sent to the application program for final processing. Self-commutative ADT operations on set elements can be handled like reduction operations. However, ADT operations on transient program variables cannot be treated like reduction operations on transient program variables. Reduction operations could be performed on local copies of a program variable and the results merged. However, this cannot in general be done for a program ADT variable; unless the ADT implementor has defined a merge operation, merging local copies cannot be performed for an ADT. An ADT operation must communicate directly with the node holding the variable (probably the node the application program is running on); unless the program variables (and not just the persistent objects) are declustered across multiple nodes, this will mandate computing the join stream in parallel and then executing some or all of the statements in the join loop at a centralized site once for each element of the join stream. This creates a potential bottleneck in the system. To recap, if a join loop contains only statements belonging to the subclass of self-commutative statements described in Chapter 4 (except for ADT operations on transient program variables), the application program does not need to need to execute the statements; the statements' execution can be initiated by the join nodes.

---

[4]We are using the term *reduction operation* somewhat broadly here to mean any `op=` statement of the form described in Chapter 4. For instance, giving an employee a $200 raise (`E->sal += 200`) qualifies.

[5]We say an object is replicated if multiple copies of it are produced to perform the join. An equi-join involving two extents will not require replication. However, some algorithms for band-joins require replicating elements of one of the sets [DEWI91]. If an object is replicated, merging the updates made to different copies of the object may be impossible.

Unfortunately, not all set loops parallelize so nicely. Some loops must be executed sequentially to avoid violating the sequential semantics of the program. Consider a database of pictures of people. Suppose there are currently two pictures of each person, and the `Picture` data structure contains two fields: `bitmap` and `sameperson`. The `Picture` set is illustrated below.

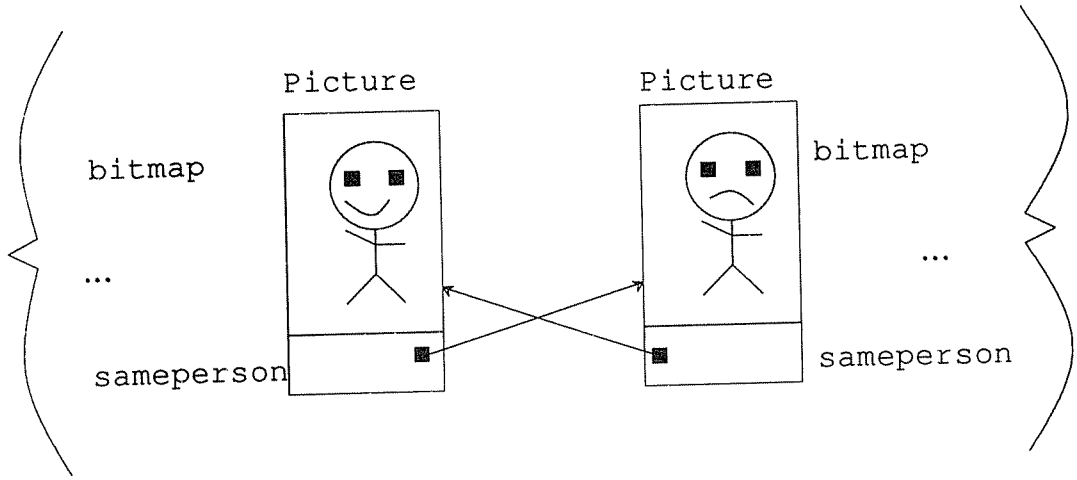

Figure 6.2: `Picture` **set**

If the query

```
(6.2) for (P of Picture)
        delete P->sameperson; //deleting a picture
```

is executed, one picture of each person should be deleted because of the sequential semantics of the language. For each pair of pictures, one will be reached first. The picture of the `sameperson` will be deleted. Thus, the second picture of the person will never be iterated over, since it will be deleted before it is reached.

Suppose the `Picture` set is declustered over multiple nodes and that one picture of *Joe* is on $node_1$ and the other is on $node_2$. If $node_1$ and $node_2$ execute `(6.2)` in parallel, they may both process *Joe*'s picture at the same time—in which case, both pictures will be deleted. Thus, there are set loops that must be executed sequentially at a central site to avoid violating program semantics.

6.2. TRANSFORMATIONS FOR PARALLELISM

As illustrated in Chapter 3, nested iterators can be used to express joins with grouping constraints. These grouping constraints can negatively impact performance by preventing the reordering of join computation, by requiring that all computation be done in the application program, and by limiting the choice of join algorithms.

Thus, it is useful to remove as many grouping constraints as possible. However, the flow of values through the program and the presence of output statements constrain the reorderings that can be performed and the join algorithms that can be used without violating the program's semantics. This section describes how the transformations of Chapter 3 can be used to facilitate the use of parallelism. Throughout this section, we will assume that predicates are side-effect free and do not involve values modified within the loop.

6.2.1. Simple Group-by Loops

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

```
(6.3) for (X1 of Set1) suchthat (Pred1(X1))
          for (X2 of Set2) suchthat (Pred2(X1,X2))
              S21;
```

If `S21` is a self-commutative statement, then, by the definition of self-commutativity, (6.3) is equivalent to:

```
(T1) for (X1 of Set1;  X2 of Set2) suchthat (Pred1(X1) && (Pred2(X1,X2))
          S21;
```

The join portion of (T1)'s result can be computed in parallel using the **Hybrid-hash** join algorithm (as the parallelization of (6.1) did). In general, the application program must perform statement `S21` for each result tuple produced by the join processors.[6] However, both the application program's workload and the amount of network traffic can be dramatically reduced if the conditions in Section 6.1 are satisfied (i.e. `S21` only contains reductions which assign to program variables, certain set insertions/deletions/modifications, and reduction operations on—or assignments of expressions that are constant for the duration of the loop to—fields of a non-replicated object participating in the join).

If `S21` does not modify `Set1`, `Set2`, or the variables used in the predicates, transformation (T2) can be used to rewrite query (6.3) as a join followed by a sort even if `S21` is not self-commutative. The parallel version of (T2) works as follows:

(1) Perform a **Hybrid-hash** join and write each result tuple to disk at the node that produced it (the result tuple must include a unique identifier, for instance an oid, for the `Set1` object that produced it).

---

[6]There may be no way to merge updates to local variables to produce the correct update to a global variable. Updates to local $\texttt{profcount}_i$'s could be merged for (6.1), but merging updates made to local copies of an ADT may not be possible.

(2) After the join at node$_i$ has completed, sort the result tuples at node$_i$ on the unique identifier for a `Set1` object which is stored in each result tuple $\forall i\ 1 \le i \le n$. When finished, node$_i$ sends a `DONE` message to the application program.

(3) As each node finishes, the sorted run of result tuples from that node are returned to the application program which performs `S21'` (a statement that looks like `S21`, except that uses of the fields of `Set1` and `Set2` are replaced by uses of the fields of `Temp`) once for each tuple.[7] The application program processes each sorted run (one run per join node) to completion before starting on the next run.

Each local sort produces a stream of result tuples that is grouped by `Set1` (e.g. if `Set1` is `Dept`, all the result tuples for the *History* department will be grouped together sequentially in the stream). Since the stream for the application program is the concatenation of the local streams, the stream in the application program will be grouped by `Set1`.

6.2.2. General Group-by Loops

The group-by loops exemplified by (6.3) 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. Chapter 3 contains transformations (T3)-(T6) for general group-by loops. All of those transformations can be used to produce better parallel code than a straightforward parallelization of the original, untransformed loop in many cases. However, we will only consider an example of how (T5) can be used for this purpose. This will suffice to give the intuition as to how to parallelize general group-by loops and how to use transformations to improve the quality of parallel code. Consider query (6.4):

```
(6.4) for (D of Dept) {
          cnt = 0;  //S11
          for (P of Professor) suchthat (D->did==P->did)
              cnt++;  //S21
          printf("%s %d ",D->name,cnt);
      }
```

Assume again that `Dept` and `Professor` are declustered across the same n nodes and that these n nodes will be used to execute the join. In addition, assume that `Dept` fits in the aggregate memory of the n processors,

[7]Since `S21` is not self-commutative, there is some order dependency. Executing `S21` on multiple result tuples simultaneously would violate the order dependency.

but that `Professor` does not. Then, without modification, query (6.4) would be executed as follows at each $node_i$ (if the **Hybrid-hash** join algorithm was chosen):

(0) Create a local variable cnt_i.

(1) Repartition `Professor`, hashing the `did` field of each `Professor` object to determine which node the object should be sent to. As a `Professor` object arrives at $node_i$, hash its `did` field using a second hash function to determine whether to insert it into the local hash table or to write it to one of N partitions on disk, $Professor_{i1}$-$Professor_{iN}$.

(2) Using the same partitioning hash function for `Dept`, redistribute the set. As a `Dept` object arrives at $node_i$, hash its `did` field using the second hash function to determine whether the object should probe the in-memory hash table of `Professor` objects or be written to one of N partitions on disk $Dept_{i1}$-$Dept_{iN}$. (Objects in partition $Professor_{ij}$ will only join with those in $Dept_{ij}$ $\forall j$ $1 \leq j \leq N$.) For each probing `Dept` object, do the following:

(a) Set cnt_i to zero.

(b) For each matching `Professor` object, increment cnt_i.

(c) Add <D->name, cnt_i> to a packet of result tuples. When this packet is full (or no more result tuples will be produced), send the packet to the application program. (The application program prints the result tuples as they arrive).

(3) Load the $Professor_{i1}$ partition into the hash table. Read $Dept_{i1}$ objects a page at a time. For each $Dept_{i1}$ object, probe the hash table, performing steps (a)-(c) of step (2). Then, process the second partition of `Professor` and `Dept`, and so on until all N partitions have been processed.

Alternatively, (T5) could be applied to query (6.4), and the resulting code could be parallelized. Applying (T5) to query (6.4) produces:

```
(6.5) Temp = [];
      for (D of Dept) {
         cnt = 0;  //S11
         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);
```

Since we assumed that Dept fits in the aggregate memory of the n processors, the parallelization of (6.5) does not require writing any Dept objects to disk if repartitioning Dept on its did field produces a relatively uniform distribution of Depts across nodes. Assuming that this is the case, query (6.5) executes at each node$_i$ as follows:

(1) Repartition Dept, hashing the did field of each Dept to determine which node the object should be sent to.[8] As Dept objects arrive at node$_i$, tag them with a new field cnt (which is set to zero) and insert them into the local hash table.

(2) Repartition Professor using the same partitioning hash function. For each arriving Professor object, probe the in-memory Dept hash table and increment the cnt field of each Dept object (if any) that it joins with.

(3) Send the name and cnt fields of each Dept object in the local hash table to the application program a packet at a time. (The application program prints them as they arrive.)

While both queries (6.4) and (6.5) can be evaluated using a straightforward modification of the parallel **Hybrid-hash** join algorithm, query (6.5) requires only a single scan of the Dept and Professor sets if each partition of Dept will fit in memory at the join processor that it is sent to. In contrast, if Professor does not fit in the aggregate memory, query (6.4) must reread part of the Professor extent no matter what join algorithm is used to evaluate the join. Thus, query (6.5) potentially incurs fewer I/Os than query (6.4).

6.3. POINTER-BASED JOIN TECHNIQUES

The transformations from Section 6.2.1 cannot be used if the simple group-by loop iterates over a set-valued attribute. Consider the following group-by loop where each element of Set1 contains a set-valued attribute named "set":

```
(6.6) for (X1 of Set1) suchthat (Pred1(X1))
        for (X2 of X1->set)  suchthat (Pred2(X1,X2))
          S21;
```

Since there are many inner sets, and not just one like in query (6.3), neither (T1) nor (T2) can be applied. This section will present execution strategies for query (6.6), but, first, some terminology is needed. If X2 is an ele-

[8] Actually, each Dept object is projected to produce a tuple, and the tuples are repartitioned.

ment of an `X1->set` instance, we term `X2` a **child** of `X1`, and `X1` a **parent** of `X2`. We term the (implicit) set of all children objects `Set2`. We do not consider the case where children have back-pointers to their parents; this eliminates a number of the execution strategies that [SHEK90, SHEK91] presented for uni-processor systems. This section describes, analyzes, and compares four execution strategies for loops like (`6.6`): the **Hash-loops** [SHEK90, MEHT91], the **Probe-children**, the **Hybrid-hash/node-pointer** [MEHT91], and the **Hybrid-hash/page-pointer** [MEHT91] join algorithms. It also contains the **Find-children** algorithm which allows algorithms like [GERB86, SCHN89, DEWI92a]'s parallel **Hybrid-hash** and our **Probe-children** join algorithm to be used even when an explicit extent of children of `Set1` does not exist. For all of the algorithms, we assume that set-valued attributes are represented as lists of oids stored inside the `Set1` objects and that the oids are "physical" [KHOS86]—that is, each oid contains information about the node and disk page of the referenced object. We term such oids **page-pointers**. Actually, only the **Hash-loops** and **Hybrid-hash/page-pointer** algorithms require **page-pointers**. **Probe-children** and **Hybrid-hash/node-pointer** only require **node-pointers**—oids neither completely physical nor completely logical, but which contain sufficient information to calculate the node that the corresponding object resides on. For example, a node pointer to a `Set2` object could be the partitioning attribute for `Set2` provided that `Set2` is an explicit extent and the partitioning attribute is a key. Thus, **Probe-children** and **Hybrid-hash/node-pointer** can be used for parallel relational database systems that support range and/or hash partitioning.

A uni-processor version of the **Hash-loops** algorithm was described and analyzed in [SHEK90] assuming no sharing of objects. [MEHT91] described a parallel version, but did not analyze it. Both implicitly assumed that all objects mentioned in the query could be accessed through an extent. We do not make this assumption, but instead propose the **Find-children** algorithm to compute an implicit extent when an explicit extent does not exist. We analyze the parallel algorithms. Our analysis can handle multiple parents sharing a child. We also develop and analyze a new pointer-based join algorithm. The spirit of the **Hash-loops** analysis was influenced by [SHEK90]. However, changes were required to account for the effects of sharing, of selection and projection, and of declustering objects. [SHEK90] did not take space overhead for a hash table into account; replacing [SHEK90]'s assumption that a hash table for c pages of persistent data requires c pages of main memory also required a number of changes.

6.3.1. Hash-loops

This algorithm is a parallel version of the **Hash-loops** join algorithm. We will first present the uni-processor version [SHEK90], and then examine a parallel version. Both require physical oids to be applicable. Let S_i be the

subset of set S at node$_i$. The uni-processor **Hash-loops** algorithm repeats the following two steps until all of the objects in `Set1` (the outer set in query (6.6)) have been processed:

(1) A memory-sized chunk of `Set1` (or the remaining portion of `Set1`, if that is smaller) is read into main memory. For each `Set1` object, the page identifier (PID) components of all the oids contained in the object's set-valued attribute are extracted and inserted into a hash table. Each hash entry contains both a PID and a list of pointers to the `Set1` objects with children on that page (e.g. in Figure 6.3, Page #20 contains child objects of both Ralph and Pat).

(2) Once the hash table is built, the hash entries are processed sequentially by reading the corresponding page into the buffer pool. Then, each `Set1` object that references the page is joined with the relevant child objects on the page.

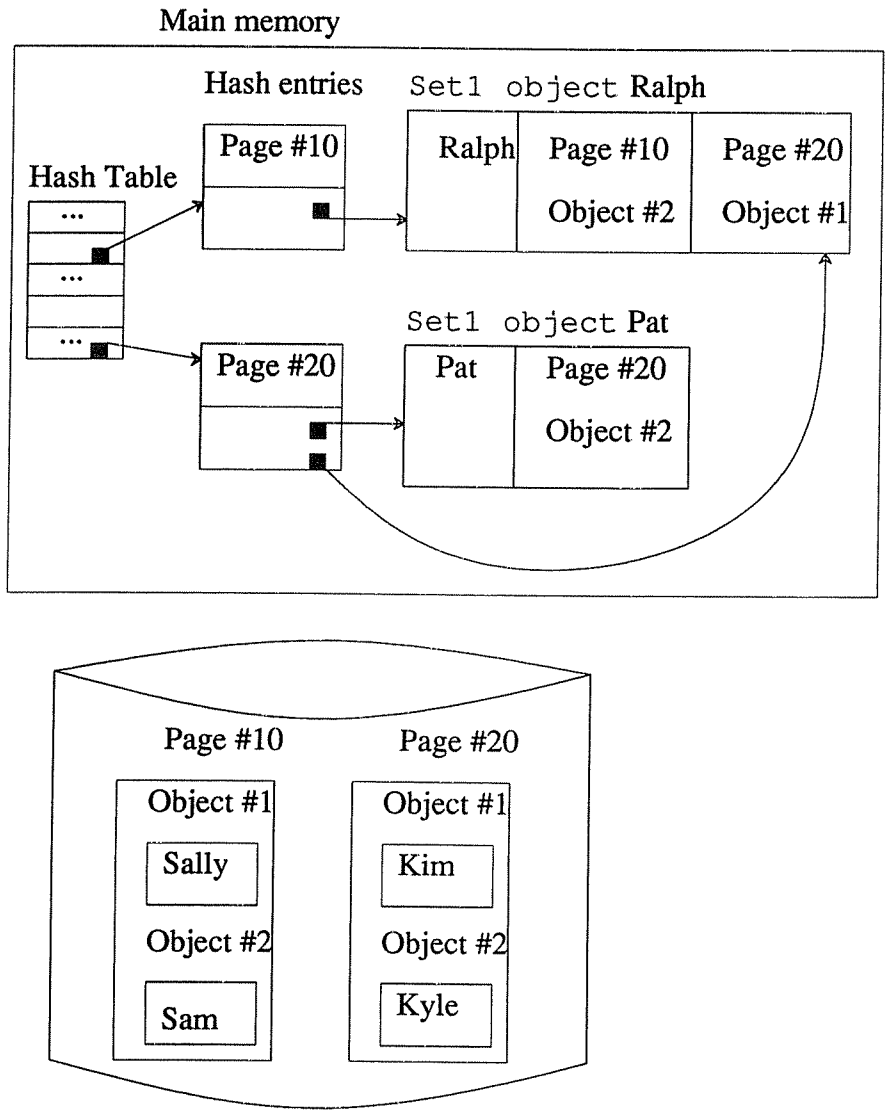

Figure 6.3: Uni-processor Hash-loops example

In Figure 6.3, each persistent pointer contains a PID and an object number. The object number is relative to the corresponding page; it is a slot identifier.

Note that this algorithm will not produce a join stream that is grouped by `Set1`. If statement `S21` is an order-dependent operation (e.g. a `printf`), the join stream will need to be partitioned *ala* (T2) (perhaps via sorting) before executing `S21`. The same comment will apply to the **Probe-children** and the two pointer-based **Hybrid-hash** join algorithms that will be presented shortly.

For example, assume that the current iteration loads two `Set1` objects into the hash table shown in Figure 6.3. Step (2) executes as follows. First, Page #10 is read from disk. The only pointer in the hash entry for Page #10 is

Ralph. Since Ralph references Sam, `S21` is executed for the pair Ralph/Sam. Next, Page #20 is read from disk. The first pointer in its hash entry is Pat which references Kyle. `S21` is executed for the pair Pat/Kyle. The second pointer in the hash entry is Ralph, and `S21` is executed for the pair Ralph/Kim.

The parallel version executes as follows at each $node_i$ $\forall i\ 1 \le i \le n$:

(1) Scan $\texttt{Set1}_i$. Project each selected $\texttt{Set1}_i$ object to produce a tuple. We will term these tuples `Set1`-tuples (to distinguish them from the original objects). A `Set1`-tuple is sent to each node that contains a child object of the original $\texttt{Set1}_i$ object. (The tuple sent to $node_j$ will only contain oids that reference $node_j$.) As `Set1`-tuples arrive at $node_i$, they are inserted into a hash table. On memory overflow, newly arriving tuples are written to a single overflow partition on disk. <Synchronization>

(2) Execute step (2) of the uni-processor algorithm.

(3) After the initial hash table is processed, repeat steps (1) and (2) of the uni-processor algorithm (with the overflow tuples taking the place of `Set1`) until there are no more overflow tuples to process.

The <Synchronization> at the end of step (1) means that no node can start step (2) until all nodes have completed step (1). Synchronization is required in previous examples as well in an implementation. We did not mention the synchronization points because they were not important to understand how the parallelization worked. However, in our analysis of algorithms, synchronization points are very important, so we represent them explicitly.

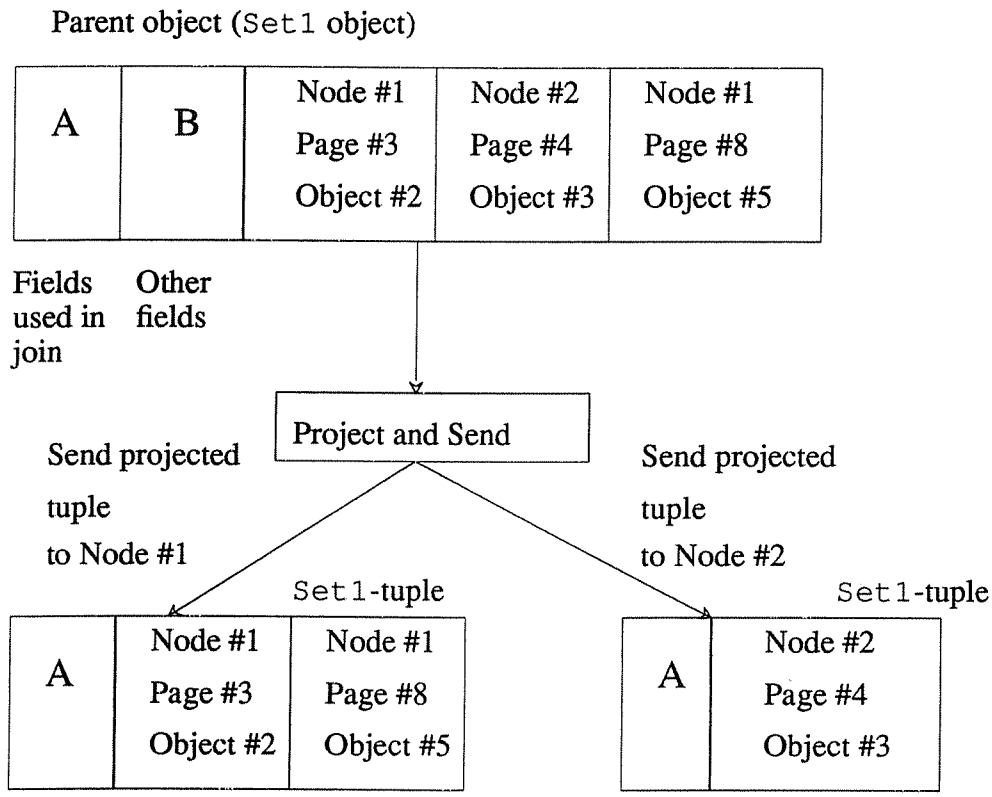

Figure 6.4: Replicating a `Set1` object during step (1) of Parallel Hash-loops

Figure 6.4 illustrates the projection and replication of a single selected `Set1` object to produce two `Set1`-tuples during step (1).

6.3.2. Find-children

If the implicit set `Set2` is not an explicit extent, one must use a parallel version of the **Hash-loops**, pointer-based nested loops, or pointer-based **Hybrid-hash** join algorithm [SHEK90]. Alternatively, an extent can be explicitly computed—which is what the **Find-children** algorithm does. **Find-children** computes which of each node's pages contain elements of the implicit set `Set2`. Given the computed extent, a standard parallel **Hybrid-hash** algorithm can be used by producing `Set1`-tuples which contain exactly one oid each and then using the oid as the join attribute. Finding the children also allows our new **Probe-children** join algorithm, described in the following section, to be applied. **Find-children** is not a join algorithm; it is an algorithm that can compute information required by certain join algorithms.

The **Find-children** algorithm proceeds as follows at each node$_i$ $\forall i\ 1 \le i \le n$:

(1) $\texttt{Set1}_i$ is scanned. Each child pointer (an oid) contained in a selected $\texttt{Set1}_i$ object is stripped of all information other than the PID component of the oid (i.e. information about the node and about the object's position on the page are removed). Each stripped pointer is put in the bucket corresponding to the node that the original pointer referenced (one of n buckets). If only some of the stripped pointers can fit in main memory, the excess stripped pointers are sent to the node that they reference. Hash tables are built locally for the memory-resident stripped pointers to make it easy to eliminate duplicates.

(2) Once $\texttt{Set1}_i$ has been processed, the memory-resident stripped pointers are sent to the appropriate nodes, where they are written to disk as they arrive. (If there is sufficient room, sorted runs of stripped pointer should be produced before writing them to disk.) <Synchronization>

(3) Pointers are sorted to remove duplicates and written back to disk at each node. (Even if step (1) eliminates all duplicate references to node_i locally at each of the n nodes, multiple nodes may reference the same page on node_i. Thus, duplicate elimination is always needed during step (3).)

Essentially, **Find-children** performs a semi-join between the pages of `Set2` and the relevant pointers of `Set1`. As a simple example, assume that two `Set1` objects are stored on node_i, each of which contains a set-valued attribute with two children, and there is sufficient room in main memory to hold all of the child pointers. Figure 6.5 illustrates the steps that the algorithm goes through up to the synchronization point. Steps (a)-(d) are from step (1) of the algorithm; steps (e) and (f) are from step (2). Stripped pointers may be received from other nodes while steps (a)-(f) are occurring, and may continue to arrive until the synchronization point is reached.

We analyzed this algorithm in some detail, and used the analysis in our algorithm comparisons in Section 6.3.7. We do not include the analysis because the cost of **Find-children** at node_i is roughly the cost of an extra scan of $\texttt{Set1}_i$ unless the number of distinct stripped pointers contained in $\texttt{Set1}_i$ objects is huge. They should all fit in main memory in most realistic cases, so there will be no need to write stripped pointers to disk during step (1). Also, the cost of sorting and writing stripped pointers to disk during step (3) is minimal compared to the cost of scanning $\texttt{Set2}_i$ for the join that uses the extent computed by **Find-children**. The results presented in this chapter include these smaller costs associated with applying **Find-children**, but the results would not be qualitatively different if only the extra scan of $\texttt{Set1}_i$ was accounted for.

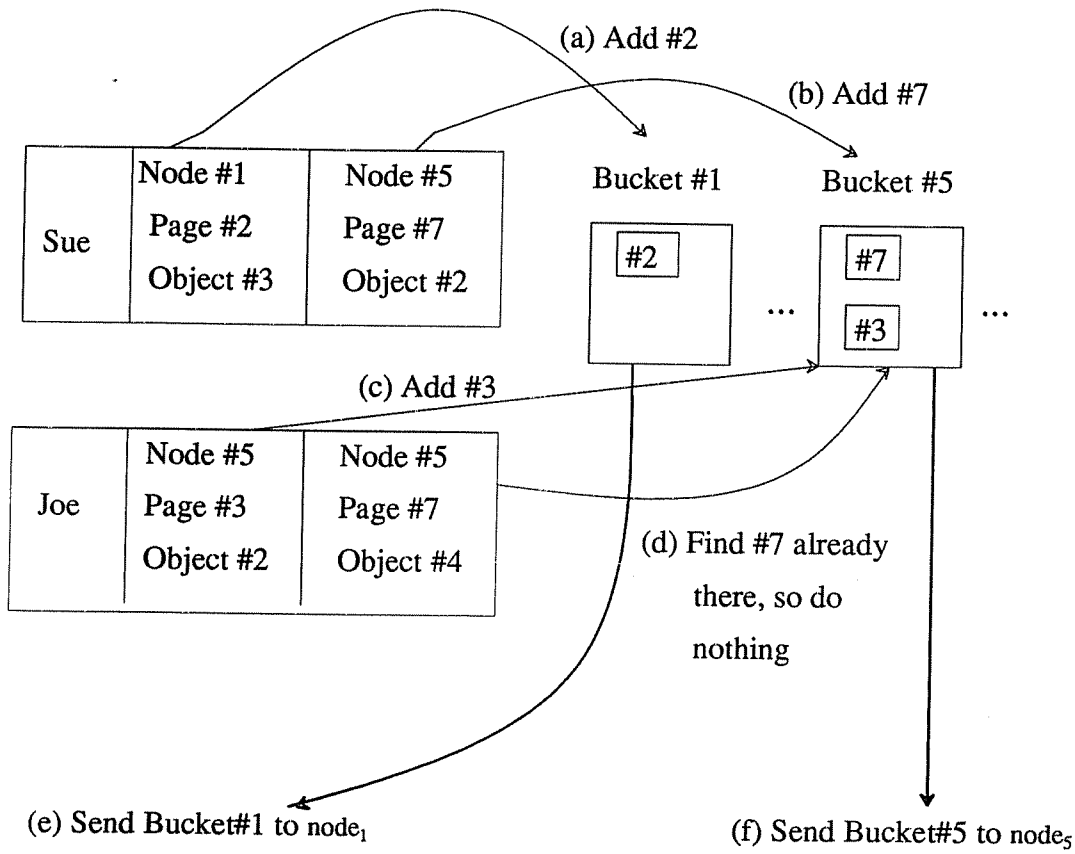

Figure 6.5: Processing two `Set1` objects by Find-children at node$_i$

6.3.3. Probe-children Join

This algorithm requires knowing the extent `Set2`. (The **Find-children** algorithm can be used to compute the extent if necessary.) **Probe-children** proceeds as follows at each node$_i$ $\forall i\ 1 \le i \le n$:

(1) `Set2`$_i$ is scanned and the selection predicate is applied. Each selected, projected `Set2`$_i$-tuple (tagged with its oid) is put into a local memory-resident hash table based on its oid. This continues until there is no more room in main memory. <Synchronization>

(2) Scan `Set1`$_i$, producing and distributing `Set1`-tuples as in step (1) of parallel **Hash-loops**. The oids contained in the set-valued attribute of each arriving `Set1`-tuple replica are hashed to find the relevant children. Produce one result tuple for each match. If all of the `Set2` objects referenced by the `Set1`-tuple are currently in the hash table, the tuple is discarded. Otherwise, the parent and unseen children oids are written to disk. <Synchronization>

(3) Repeat the following two steps until all of $Set2_i$ has been processed.

(a) Scan the unread portion of $Set2_i$ and put the selected $Set2_i$-tuples (tagged with their oids) into the hash table until the hash table fills or the set is exhausted.

(b) Read the parents and probe the hash table for children.[9]

Here is an example (using data from the example in the **Find-children** section) of processing a single Set1-tuple at $node_5$.

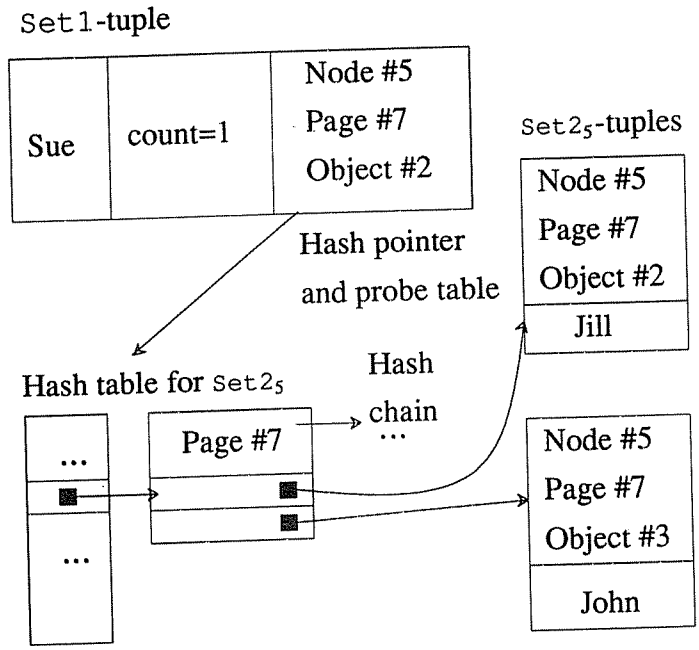

Figure 6.6: Processing a Set1-tuple by Probe-children at $node_5$

For simplicity, we use the page identifier as the hash value of a pointer in this example—in general a more complicated hash function will be used. The object's pointer is hashed and the hash entry for Page #7 is found. The hash entry's pointers to $Set2_5$-tuples are followed and the oids in the hash table's tuples are compared to the oid in Sue's set-valued attribute. Jill matches, so S21 is executed for the pair Sue/Jill. If the match was found during

[9]A variant of this algorithm would only keep parents that have unseen children. This requires a write of such parents during step (b). The variant has some similarities to the **Simple-Hash** join algorithm [DEWI84]. In both, the tuples that need to be processed later are written to a new file and that new file is used for the next iteration. The actual **Probe-child** join algorithm, however, proceeds in much the same way as the **Hashed Loops** join algorithm for centralized databases [GERB86]. Both build a memory-sized hash table for some fraction of the inner set Set2. They read the whole (local partition of the) set of Set1-tuples to probe the hash table. This continues until all of Set2 has been processed. There are two major differences between the uni-processor **Probe-children** and **Hashed Loops**. First, **Probe-children** probes the hash table once for each pointer in its set-valued attribute; **Hashed Loops** probes the hash table exactly once. Second, **Probe-children** eliminates some Set1-tuples during step (2)—much like **Simple-Hash**; **Hashed Loops** reads all of Set1 once for each hash table.

step (2), Sue need not be written to disk for processing during step (3) as there will be no more $\texttt{Set2}_5$-tuples that Sue will join with.

6.3.4. Hybrid-hash/node-pointer

Hybrid-hash/node-pointer, like **Probe-children**, requires knowing the extent `Set2`. The algorithm proceeds as follows at each node_i $\forall i\ 1 \le i \le n$:

(1) $\texttt{Set2}_i$ is scanned and the selection predicate is applied. Each selected, projected $\texttt{Set2}_i$-tuple (tagged with its oid) is hashed on its oid and inserted into one of $B+1$ buckets. Tuples in the first bucket are inserted into a main memory hash table. All others are written to disk. The value of B is chosen such that (1) the 1st bucket of `Set1`-tuples can be joined in memory with the first bucket of $\texttt{Set2}_i$-tuples during step (2), and (2) the hash table for the j-th $\texttt{Set2}_i$ bucket can fit in main memory $\forall j\ 2 \le j \le B+1$. <Synchronization>

(2) Scan $\texttt{Set1}_i$ and distribute `Set1`-tuples, each of which contains exactly one `Set2` (child) oid, to the relevant `Set2` node. As a tuple arrives, its child oid is hashed and the tuple is classified into one of $B+1$ buckets. The j-th `Set1` bucket will only join with the j-th $\texttt{Set2}_i$ bucket. The `Set1`-tuples of the first bucket probe the hash table; the other `Set1` elements are written to disk.

(3) The next $\texttt{Set2}_i$ bucket is loaded into the hash table, and the tuples of the corresponding `Set1` bucket probe it. This continues until all the buckets have been processed.

This algorithm is very similar to **Probe-child**. There are three main differences. First, **Hybrid-hash/node-pointer** may write and reread part of `Set2`; **Probe-children** will only read `Set2` once. Second, **Probe-children** produces one replica of a `Set1` object per referenced node; **Hybrid-hash/node-pointer** produces one replica per pointer. Thus, the **Probe-children** algorithm will potentially produce fewer `Set1`-tuples. Third, **Probe-children** may reread the same `Set1`-tuples multiple times; **Hybrid-hash/node-pointer** will reread `Set1`-tuples at most once because it partitions `Set1`- and `Set2`- tuples into buckets.

6.3.5. Hybrid-hash/page-pointer

This algorithm is almost identical to the pointer-based **Hybrid-hash** join algorithm of [SHEK90]. Only step (1) which redistributes `Set1` is different. The algorithm proceeds as follows at each node_i $\forall i\ 1 \le i \le n$:

(1) Scan `Set1`$_i$ and distribute `Set1`-tuples, each of which contains exactly one `Set2` (child) oid, to the relevant `Set2` node. As a tuple arrives, the PID component of its child oid is hashed and the tuple is put into one of B+1 buckets. Tuples in the first bucket are inserted into a main memory hash table. All others are written to disk. <Synchronization>

(2) Execute step (2) of the uni-processor **Hash-loops** join algorithm.

(3) After the initial hash table is processed, load the next `Set1` bucket into the hash table. Process the table using step (2) of the uni-processor **Hash-loops** join algorithm. This continues until all the buckets have been processed.

The only differences between this algorithm and **Hash-loops** are that (1) each tuple has only one pointer, and (2) each `Set2` page is read only once (because of partitioning tuples into buckets).

6.3.6. Analysis of Pointer-based Join Algorithms

This section analyses the performance of the four join algorithms described in Sections 6.3.1 through 6.3.5 for queries like:

```
(6.7) for (X1 of Set1) suchthat (Pred1(X1))
        for (X2 of X1->set)  suchthat (Pred2(X1,X2))
          S21;
```

To simplify our analysis, we assume that I/O is only performed on behalf of the join—that `S21` does not perform any I/O.

6.3.6.1. Assumptions for Analysis

We assume that the selection predicates on `Set1` and `Set2` are equally selective at each node (i.e. no Selectivity Skew [WALT91]). We repeat the definitions of two functions from Chapter 3 for use in the analysis. The first is $\delta(s,p)$, the number of objects of size s that fit on a page of size p:

$$\delta(s,p) = \left\lfloor \frac{p}{s} \right\rfloor$$

The second is $\theta(m,s,p)$, the number of pages of size p required to hold m objects of size s:

$$\theta(m,s,p) = \left\lceil \frac{m}{\delta(s,p)} \right\rceil$$

Table 6.1 shows the system parameters and Table 6.2 shows the catalog information used by the analysis in later

sections.

Name	Description	Default
n	number of nodes	32
P	size of a disk page	8192 bytes
M_i	number of memory buffer pages at node$_i$	varied
IO	time to read or write a page	20 msec
$size_{ptr}$	size in bytes of a persistent pointer	12 bytes
F	hash table for m pages of data requires $F \cdot m$ pages	1.2
k	each `Set1` object has k children	10
f	each `Set2` object has f parents	2

Table 6.1: System Parameters

Name	Description
$\|S_i\|$	number of objects in set S_i
$size_S$	size in bytes of an object in set S
$\pi width_S$	size in bytes of projected S-tuple (not including size of the set-valued attribute, if any, in S-tuple)
sel_S	selectivity of selection predicate on S [10]
α_{ij}	fraction of pointers in `Set1`$_i$ objects that point to `Set2`$_j$ objects
ϕ^x_{ij}	number of `Set1`$_i$ objects that reference `Set2`$_j$ objects
Ξ_{S_i}	cardinality of the selected subset of S_i, $\Xi_{S_i} = \|S_i\| \cdot sel_S$ since the selection predicates are assumed to be equally selective at each node
ϕ_{ij}	number of selected `Set1`$_i$ objects that reference `Set2`$_j$ objects, $\phi_{ij} = \phi^x_{ij} \cdot sel_{\texttt{Set1}}$
ρ_{ij}	number of pointers from selected `Set1`$_i$ objects that reference `Set2`$_j$ objects, $\rho_{ij} = \Xi_{Set1_i} \cdot k \cdot \alpha_{ij}$
O_S	number of S objects per page, $O_S = \delta(size_S, P)$
P_{S_i}	number of pages of S_i $P_{S_i} = \theta(\|S_i\|, size_S, P)$

Table 6.2: Catalog Information

The assumption that each object of Set1 has k children and that each child has f parents implies that each Set2 object is equally likely to be referenced from a randomly chosen Set1 object. Also, dividing the number of pointers that reference $Set2_i$ by f gives the cardinality of $Set2_i$. Thus, $|Set2_i| = \frac{\sum_{j=1}^{n}(|Set1_j| \cdot \alpha_{ji} \cdot k)}{f}$.

Table 6.2 implicitly assumes that each data page contains only Set1 or Set2 objects. We also assume that the α_{ij} are the same for the selected subset of $Set1_i$ as for all of $Set1_i$ (this assumption shows up in the formula for ρ_{ij} in Table 6.2). Since our algorithm comparisons do not select Set1, this assumption does not affect our results.

We do not include CPU times in our analysis; a similar join algorithm analysis in [WALT91] included CPU time and found that none of the queries were CPU bound—the CPU time was always lost in the overlap with I/O and communication. Thus, we feel it is safe to neglect CPU time in our analysis. Originally, we used [WALT91]'s style of capturing overlapped communication and I/O. We removed communication from this analysis for two reasons. First, it is easy to derive the number of messages sent using the analysis of I/Os performed. Second, communication was completely overlapped with I/O in all our algorithm comparisons (which assumed that sending a 8K message takes 5 msec and performing an I/O with an 8K page takes 20 msec). Communication will only become the dominant cost for the algorithms considered in this section if each object must be replicated many times and each node has enough main memory to hold all of its hash table data in a single hash table.

6.3.6.2. Analysis of Hash-loops

Before we can estimate the number of I/Os performed, we must estimate several other quantities. During step (1), $node_i$ is expected to receive

$$\phi_i^r = \sum_{j=1}^{n} \phi_{ji}$$

Set1-tuples and $\sum_{j=1}^{n} \rho_{ji}$ pointers. Thus, the average number of pointers per Set1-tuple received at $node_i$ is

$$a_i^r = \frac{\rho}{\phi_i^r} \text{ where } \rho = \sum_{j=1}^{n} \rho_{ji}$$

---

[10] Sel_S is the selectivity of *SelectPred* in Chapter 5's node representation of the S set loop.

The tuples received at node$_i$ will be $\pi width_{\texttt{Set1}}$ bytes long from the projected fields other than the set-valued attribute. The set-valued attribute can be represented with an integer count and a list of oids, so the average length of a `Set1`-tuple will be

$$\pi ave_i^r = \pi width_{\texttt{Set1}} + \texttt{sizeof(int)} + ar_i \cdot size_{ptr}$$

Thus, node$_i$ will receive approximately

$$PagesReceived_i^{\pi\sigma} = \theta(\phi r_i, \pi ave_i^r, P)$$

pages of selected, projected `Set1`-tuples, each of which contains approximately

$$TuplesPerPage_i^{HL} = \delta(\pi ave_i^r, P)$$

tuples. During step (1), node$_i$ will need one input buffer for reading the `Set1`$_i$ objects it must distribute to the n nodes, n output buffers for sending (replicated) `Set1`-tuples, one input buffer to receive incoming `Set1`-tuples from other nodes, and one output buffer for writing overflow `Set1`-tuples to disk. Thus, node$_i$ will have $M_i - (n+3)$ pages available for its hash table. Since a hash table for c pages of data is assumed to take $c \cdot F$ pages of main memory,

$$PagesInMem_i^{HL_1} = \min\left(PagesReceived_i^{\pi\sigma}, \left\lfloor \frac{M_i - (n+3)}{F} \right\rfloor\right)$$

pages of `Set1`-tuples can be put into the hash table at node$_i$, and

$$ToDisk_i^{HL} = PagesReceived_i^{\pi\sigma} - PagesInMem_i^{HL_1}$$

pages must be written to disk for processing during step (3). Thus, node$_i$ is expected to perform the following number of I/Os during step (1):

$$T_{IO_i}^{HL_1} = \begin{array}{ll} P_{\texttt{Set1}_i} & \text{read } \texttt{Set1}_i \\ +ToDisk_i^{HL} & \text{write overflow pages of } \texttt{Set1}\text{-tuples to disk} \end{array}$$

To estimate the amount of work done by node$_i$ during step (2), note that a given `Set2`$_i$ page will be read at most one time per hash table. To calculate the number of page reads for `Set2`$_i$ objects, we use a formula from [YAO77] for calculating the expected fraction of pages of a set with cardinality $|S|$ that must be read to examine a subset of size $|S_\sigma|$ provided O_S objects fit on a page. Yao's formula is:

$$Y(|S|,O_S,|S_\sigma|) = 1 - \frac{\binom{|S|-O_S}{|S_\sigma|}}{\binom{|S|}{|S_\sigma|}} = 1 - \prod_{i=1}^{O_s} \frac{|S|-|S_\sigma|-i+1}{|S|-i+1}$$

To use Yao's formula to calculate the fraction of $\texttt{Set2}_i$ pages that must be read for a particular hash table, an estimate of the number of $\texttt{Set2}_i$ objects that are referenced by r Set1 objects is needed. To estimate this quantity, we used combinatorics to derive *Obj(r,z,c)*, the expected number of $\texttt{Set}_{\texttt{in}}$ objects referenced by r $\texttt{Set}_{\texttt{out}}$ objects where each $\texttt{Set}_{\texttt{out}}$ object contains a set-valued attribute with an average of z pointers to $\texttt{Set}_{\texttt{in}}$ objects ($|\texttt{Set}_{\texttt{in}}|=c$). We derived $Obj(r,z,c) = c \cdot \left[1-\left[1-\frac{z}{c}\right]^r\right]$, but found that replacing it with the much simpler approximation $Obj(r,z,c) = \min(\lceil r \cdot z \rceil, c)$ produced nearly identical timing results.

We use this formula to estimate the number of I/Os performed during step (2). The $(PagesInMem_i^{HL_1} \cdot TuplesPerPage_i^{HL})$ hash table entries of step (2) will reference approximately $Obj((PagesInMem_i^{HL_1} \cdot TuplesPerPage_i^{HL}), ar_i, |\texttt{Set2}_i|)$ $\texttt{Set2}_i$ objects. Using Yao's formula with this estimated $\texttt{Set2}_i$ subset size, the fraction of $\texttt{Set2}_i$ pages that must be accessed is approximately $\texttt{Y}(|\texttt{Set2}_i|, O_{\texttt{Set2}}, Obj((PagesInMem_i^{HL_1} \cdot TuplesPerPage_i^{HL}), ar_i, |\texttt{Set2}_i|))$. Thus, step (2) is expected to perform

$$Step2_i = P_{\texttt{Set2}_i} \cdot \texttt{Y}(|\texttt{Set2}_i|, O_{\texttt{Set2}}, Obj((PagesInMem_i^{HL_1} \cdot TuplesPerPage_i^{HL}), ar_i, |\texttt{Set2}_i|))$$

$\texttt{Set2}_i$ page reads.

To estimate the number of $\texttt{Set2}_i$ reads that must be performed during step (3) to process the $ToDisk_i^{HL}$ pages written to disk during step (1), we use reasoning similar to the above. Step (3) at node$_i$ should perform

$$Step3_i = \begin{cases} (N_i-1) \cdot P_{\texttt{Set2}_i} \cdot \texttt{Y}(|\texttt{Set2}_i|, O_{\texttt{Set2}}, Obj(x_i, ar_i, |Set2_i|)) \\ + P_{\texttt{Set2}_i} \cdot \texttt{Y}(|\texttt{Set2}_i|, O_{\texttt{Set2}}, Obj(y_i, ar_i, |Set2_i|)) & \text{if } ToDisk_i^{HL} > 0 \\ 0 & \text{otherwise} \end{cases}$$

$\texttt{Set2}_i$ reads where

$o_i = ToDisk_i^{HL} \cdot TuplesPerPage_i^{HL}$ number of Set1-tuples to process

$x_i = \left\lfloor \frac{M_i - 1}{F} \right\rfloor \cdot TuplesPerPage_i^{HL}$ number of Set1-tuples that fit in a main memory hash table—one page needed for reading $Set2_i$ objects

$N_i = \left\lceil \frac{o_i}{x_i} \right\rceil$ number of **Hash-loops** iterations at node$_i$ for step (3)

$y_i = o_i - (N_i - 1) \cdot x_i$ number of Set1-tuples for the N_i-th iteration

Thus, node$_i$ should perform the following number of I/Os during steps (2) and (3):

$$T_{IO_i}^{HL_{2,3}} = \begin{array}{ll} Step2_i & \text{read } Set2_i \text{ pages to process hash table of step (2)} \\ +ToDisk_i^{HL} & \text{read } ToDisk_i^{HL} \text{ pages of Set1-tuples during step (3)} \\ +Step3_i & \text{read } Set2_i \text{ pages to process hash tables of step (3)} \end{array}$$

To compute the expected run time, we must add the the length of time spent getting to each synchronization point (i.e. to the end of step (1) and then to the end of step (3)). Thus the expected run-time for parallel **Hash-loops** is:

$$\left[\max\left\{ T_{IO_i}^{HL_1} \mid i \in \{1,\ldots,n\} \right\} + \max\left\{ T_{IO_i}^{HL_{2,3}} \mid i \in \{1,\ldots,n\} \right\} \right] \cdot IO$$

6.3.6.3. Analysis of Probe-children Join

There is only one difference between applying **Probe-children** with an explicit and a computed extent. In the second case, the stripped pointers must be read from disk and one buffer page must be reserved for this purpose. This will not make any qualitative difference, so we will ignore it in the analysis that follows in this paper. The results presented in this chapter do include these smaller costs, but the results would not be qualitatively different if these costs were ignored.

At most $(M_i - (n+3))$ pages can be devoted to the hash table loaded in step (1), since during step (2), one buffer will be needed for reading $Set1_i$ objects, n buffers for sending $Set1_i$-tuples, one buffer for receiving Set1-tuples from other nodes, and one buffer for writing Set1-tuples to disk. There are Ξ_{Set2_i} $Set2_i$-tuples that will need to be loaded into a hash table during steps (1) and (3). Each of these tuples will need to be tagged by its oid, so each hash table entry will be $(\pi width_{Set2} + size_{ptr})$ bytes long and

$$O_{ptr}^{Set2} = \delta\left((\pi width_{Set2} + size_{ptr}), P \right)$$

entries will fit on a page. Thus step (1) will load

$$TupsInMem_i^{PC1} = \min(\Xi_{Set2_i}, v) \text{ where } v = \left\lfloor \frac{M_i - (n+3)}{F} \right\rfloor \cdot O_{ptr}^{Set2}$$

tuples into the hash table at node$_i$. Assuming the selected Set2 objects are uniformly distributed across pages of $Set2_i$,

$$T_{IO_i}^{PC1} = \frac{TupsInMem_i^{PC1}}{\Xi_{Set2_i}} \cdot P_{Set2_i}$$

$Set2_i$ pages will be read during step (1). (This assumption is made to fairly divide up the cost of reading $Set2_i$ among steps (1) and (3); the same expected run-time would be obtained for the algorithm if the complete cost of the read was charged to either step (1) or step (3) alone.)

During step (2), if all of the $Set2_i$-tuples fit in main memory (i.e. $TupsInMem_i^{PC1}=\Xi_{Set2_i}$), then no Set1-tuples need to be written to disk; otherwise, some fraction of them must. In this case, we will make the worst case assumption that all $PagesReceived_i^{\pi\sigma}$ pages of Set1-tuples will be written to disk.[11] Thus, node$_i$ will perform the following number of I/Os during step (2):

$$T_{IO_i}^{PC2} = P_{Set1_i} + \begin{cases} 0 & \text{if } TupsInMem_i^{PC1}=\Xi_{Set2_i} \\ PagesReceived_i^{\pi\sigma} & \text{otherwise} \end{cases}$$

read local pages of Set1 during step (2)

write Set1 pages to disk during step (2)

During step (3), there will be $(\Xi_{Set2_i} - TupsInMem_i^{PC1})$ $Set2_i$-tuples to load into a hash table. During each iteration of step (3), one page must be reserved for reading Set1-tuples, so $(M_i - 1)$ pages are available for a hash table. Thus,

$$TuplesProcessed_i^{PC3} = \left\lfloor \frac{M_i - 1}{F} \right\rfloor \cdot O_{ptr}^{Set2}$$

Set2-tuples can be processed in each step (3) iteration. It follows that

[11] $PagesReceived_i^{\pi\sigma}$ is the number of pages of selected, projected Set1-tuples that reference $Set2_i$—see analysis of **Hash-loops** for the actual formula.

$$Iterations_i^{PC3} = \left\lceil \frac{\Xi_{Set2_i} - TupsInMem_i^{PC1}}{TuplesProcessed_i^{PC3}} \right\rceil$$

iterations of step (3) will be required—each of which will require reading all $PagesReceived_i^{\pi\sigma}$ pages of `Set1`-tuples. Thus, node$_i$ should perform the following number of I/Os during step (3):

$$T_{IO_i}^{PC3} = Iterations_i^{PC3} \cdot PagesReceived_i^{\pi\sigma}$$ read pages of `Set1` written during step (2) once for each iteration of step (3)

$$+ \frac{\Xi_{Set2_i} - TupsInMem_i^{PC1}}{\Xi_{Set2_i}} \cdot P_{Set2_i}$$ read `Set2`$_i$ pages during step (3)—P_{Set2_i} `Set2`$_i$ pages read during steps (1) and (3)

Since synchronization is required after each step, the expected run-time for **Probe-children** is:

$$\left[\max\left\{ T_{IO_i}^{PC1} \mid i \in \{1,...,n\} \right\} + \max\left\{ T_{IO_i}^{PC2} \mid i \in \{1,...,n\} \right\} + \max\left\{ T_{IO_i}^{PC3} \mid i \in \{1,...,n\} \right\} \right] \cdot IO$$

6.3.6.4. Analysis of Hybrid-hash/node-pointer

As in our analysis of **Probe-children**, we will ignore the minor differences between using an explicit versus a computed extent. During step (1), selected, projected `Set2`$_i$-tuples will be produced. Since each tuple is tagged with the original object's oid, the tuples will require

$$P_i^{HH-H} = \theta\left(\Xi_{Set2_i}, (\pi width_{Set2} + size_{ptr}), P\right)$$

pages of storage. During step (2), one page is required for reading `Set1`$_i$ objects, n pages are required for sending `Set1`-tuples, and one page is required for receiving tuples from other nodes. Thus,

$$M_i^{HH} = M_i - (n+2)$$

pages are available for the hash table and the output buffers at node$_i$. Using reasoning from [DEWI84],

$$B_i^{HH-H} = \left\lceil \frac{P_i^{HH-H} \cdot F - M_i^{HH}}{M_i^{HH} - 1} \right\rceil$$

output buffers are needed and the fraction

$$q_i^{HH-H} = \min\left(1.0, \frac{M_i^{HH} - B_i^{HH-H}}{P_i^{HH-H} \cdot F}\right)$$

of the pages of $\texttt{Set2}_i$-tuples can be put in the hash table (the min is required because at most 100% can be put into the hash table). Thus,

$$Overflow_i^{H1} = \left\lceil P_i^{HH-H} \cdot (1 - q_i^{HH-H}) \right\rceil$$

pages of $\texttt{Set2}_i$-tuples must be written to disk, and step (1) will require the following number of I/Os:

$$T_{IO_i}^{HH-H_1} = \begin{array}{ll} P_{\texttt{Set2}_i} & \text{read } \texttt{Set2}_i \text{ to select and project} \\ +Overflow_i^{H1} & \text{write overflow } \texttt{Set2}_i\text{-tuples} \end{array}$$

During step (2), $\texttt{Set1}_i$ must be read. Node_i will receive $\sum_{j=1}^{n} \rho_{ji}$ Set1-tuples. Since each tuple contains exactly one pointer, these tuples will require

$$P_i^{HH} = \theta\left(\sum_{j=1}^{n} \rho_{ji}, (\pi width_{\texttt{Set1}} + size_{ptr}), P\right)$$

pages of memory, and

$$Overflow_i^{H2} = \left\lceil P_i^{HH} \cdot (1 - q_i^{HH-H}) \right\rceil$$

pages of these tuples must be written to disk. The other tuples probe the first hash table. Thus, step (2) must perform:

$$T_{IO_i}^{HH-H_2} = \begin{array}{ll} P_{\texttt{Set1}_i} & \text{read } \texttt{Set1}_i \\ +Overflow_i^{H2} & \text{write overflow Set1-tuples} \end{array}$$

I/Os. Step (3) must read the partitions written to disk, so it must perform

$$T_{IO_i}^{HH-H_3} = \begin{array}{ll} Overflow_i^{H1} & \text{read overflow } \texttt{Set2}_i\text{-tuples} \\ +Overflow_i^{H2} & \text{read overflow Set1-tuples} \end{array}$$

I/Os. The only synchronization required is at the end of step (1), so the expected **Hybrid-hash/node-pointer** runtime is:

$$\left\lceil \max\left\{ T^{HH-H_1}_{IO_i} \mid i \in \{1,\ldots,n\} \right\} + \max\left\{ T^{HH-H_2}_{IO_i} + T^{HH-H_3}_{IO_i} \mid i \in \{1,\ldots,n\} \right\} \right\rceil \cdot IO$$

6.3.6.5. Analysis of Hybrid-hash/page-pointer

Using the analysis of **Hybrid-hash/node-pointer**, we estimate that node$_i$ will receive P_i^{HH} pages of `Set1`-tuples. Since the space requirements of step (1) of **Hybrid-hash/page-pointer** and step (2) of **Hybrid-hash/node-pointer** for repartitioning `Set1` are identical, M_i^{HH} pages are available for the hash table and the output buffers at node$_i$. Using analysis in [DEWI84],

$$B_i^{HH-P} = \left\lceil \frac{P_i^{HH} \cdot F - M_i^{HH}}{M_i^{HH} - 1} \right\rceil$$

output buffers will be needed, and the fraction

$$q_i^{HH-P} = \min\left[1.0, \frac{M_i^{HH} - B_i^{HH-P}}{P_i^{HH} \cdot F} \right]$$

of the arriving pages of `Set1`-tuples can be put in the hash table. Thus,

$$Overflow_i^{P1} = \left\lceil P_i^{HH} \cdot (1 - q_i^{HH-P}) \right\rceil$$

pages must be written to disk, and the cost of step (1) is:

$T^{HH_1}_{IO_i} =$	$P_{\texttt{Set1}_i}$	read `Set1`$_i$ to select, project, and replicate
	$+Overflow_i^{P1}$	write overflow buffers to disk.

Since partitioning is on the PID component of an oid, no `Set2` pages will be reread. Thus, steps (2) and (3) will require the following number of I/Os.

$T^{HH_{2,3}}_{IO_i} =$	$Overflow_i^{P1}$	read overflow buffers
	$+P_{\texttt{Set2}_i}$	read `Set2` pages

This will overestimate the number of `Set2`$_i$ pages read if the selection predicate on `Set1` is very restrictive, since in this case some `Set2`$_i$ pages will not need to be read at all. However, since our algorithm comparisons involve selecting all of `Set1`, such a correction would not affect the results. Thus, the expected run-time of **Hybrid-hash/page-pointer** is:

$$\left\lceil \max\left\{ T^{HH_1}_{IO_i} \mid i \in \{1,\ldots,n\} \right\} + \max\left\{ T^{HH_{2,3}}_{IO_i} \mid i \in \{1,\ldots,n\} \right\} \right\rceil \cdot IO$$

6.3.7. Comparison of the Algorithms for Set-Valued Attributes

In this section, we will compare the four algorithms analyzed in Section 6.3.6. We will call **Hybrid-hash/node-pointer** and **Probe-children** the **load-child** algorithms; **Hybrid-hash/page-pointer** and **Hash-loops** the **load-parent** algorithms; and **Probe-children** and **Hash-loops** the **low-replication** algorithms (because they produce one replica per node rather than one per pointer). We assume in all of our comparisons that the selection predicates are equally selective at each node (i.e. no Selectivity Skew [WALT91]).

6.3.7.1. Poorly Clustered Database

In the first algorithm comparison, the system defaults from Table 6.1 were used. Data was uniformly distributed across n=32 nodes, with $|\texttt{Set1}_i|$=6080 and $|\texttt{Set2}_i|$=30,400 $\forall i\ 1 \le i \le n$. Each `Set1` object had k=10 children, and each `Set2` object had f=2 parents. Also, each `Set1` object was $\texttt{size}_{\texttt{Set1}}$=(256+`sizeof(int)`+$k \cdot size_{ptr}$)=380 bytes long, and each `Set2` object was $\texttt{size}_{\texttt{Set2}}$=256 bytes long. Projected `Set1`-tuples were 128 bytes for fields other than the set-valued attribute (which contained one or more pointers). Projected `Set2`-tuples were exactly 128 bytes long. The average number of pointers per object received at node$_i$ for the **low-replication** algorithms, ar_i, was set to 1.15 (the number that would be expected if no clustering of references was attempted, and pointers from objects at node$_i$ are randomly distributed across the 32 nodes). M_i (the number of memory buffer pages at each node$_i$) and $sel_{\texttt{Set2}}$ (the selectivity of the predicate on `Set2`) were both individually varied; the unvaried parameter's value appears at the top of the graph describing the results. All the objects of `Set1` are selected. `Set2` does not exist as an explicit extent, so the **load-child** algorithms (i.e. **Hybrid-hash/node-pointer** and **Probe-children**) algorithms must compute it using **Find-children.**

In Figure 6.7, we compare the four algorithms across a range of memory allocations. **Probe-children** is a step function because node$_i$ must reread the replicated `Set1`-tuples sent to it once for each $\texttt{Set2}_i$ hash table. The analysis actually over-estimates both the cost of the algorithm and the size of the step because it assumes that no replicated `Set1`-tuples are eliminated during step (2), although some almost certainly are. Also, in reality the **load-child** algorithms should perform better relative to the **load-parent** algorithms than the graph indicates because the **load-child** algorithms read `Set2` pages sequentially (since **Find-children** sorts the page identifiers) while the **load-parent** algorithms read them randomly. However, following [SHAP86, WALT91], our analysis did not take

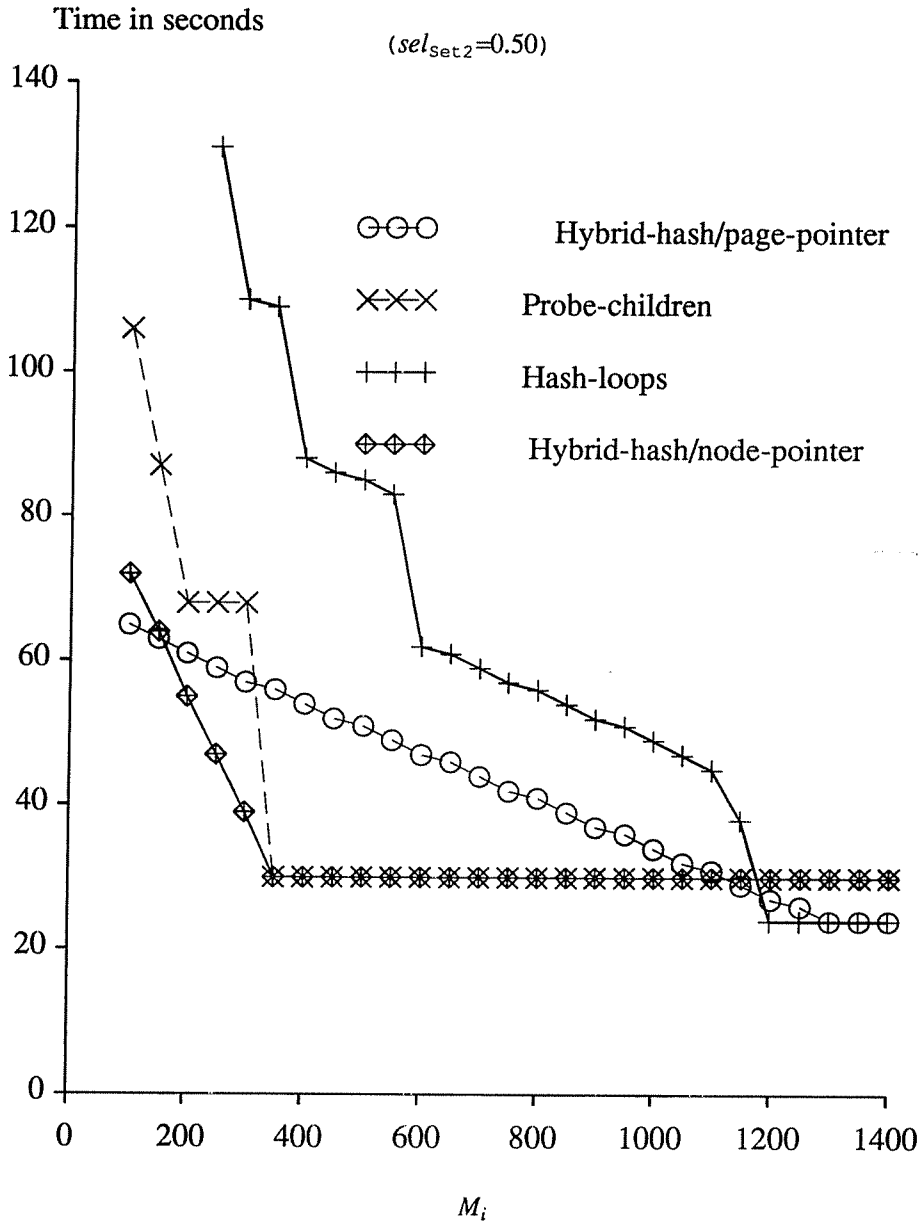

Figure 6.7: Poorly Clustered Database

the different types of I/O into account.

Hybrid-hash/node-pointer outperforms **Probe-children** at low memory sizes because it must perform I/Os for only a fraction of the `Set1`-tuples sent to each node, while **Probe-children** must write and read them all (several times). The **load-child** (i.e. **Hybrid-hash/node-pointer** and **Probe-children**) have the best performance at moderate memory sizes (where all the `Set2`-tuples fit in a main memory hash table). Since the number of selected $\texttt{Set2}_i$ objects is considerably less than the number of replicated `Set1`-tuples sent to node_i, the hash tables for the

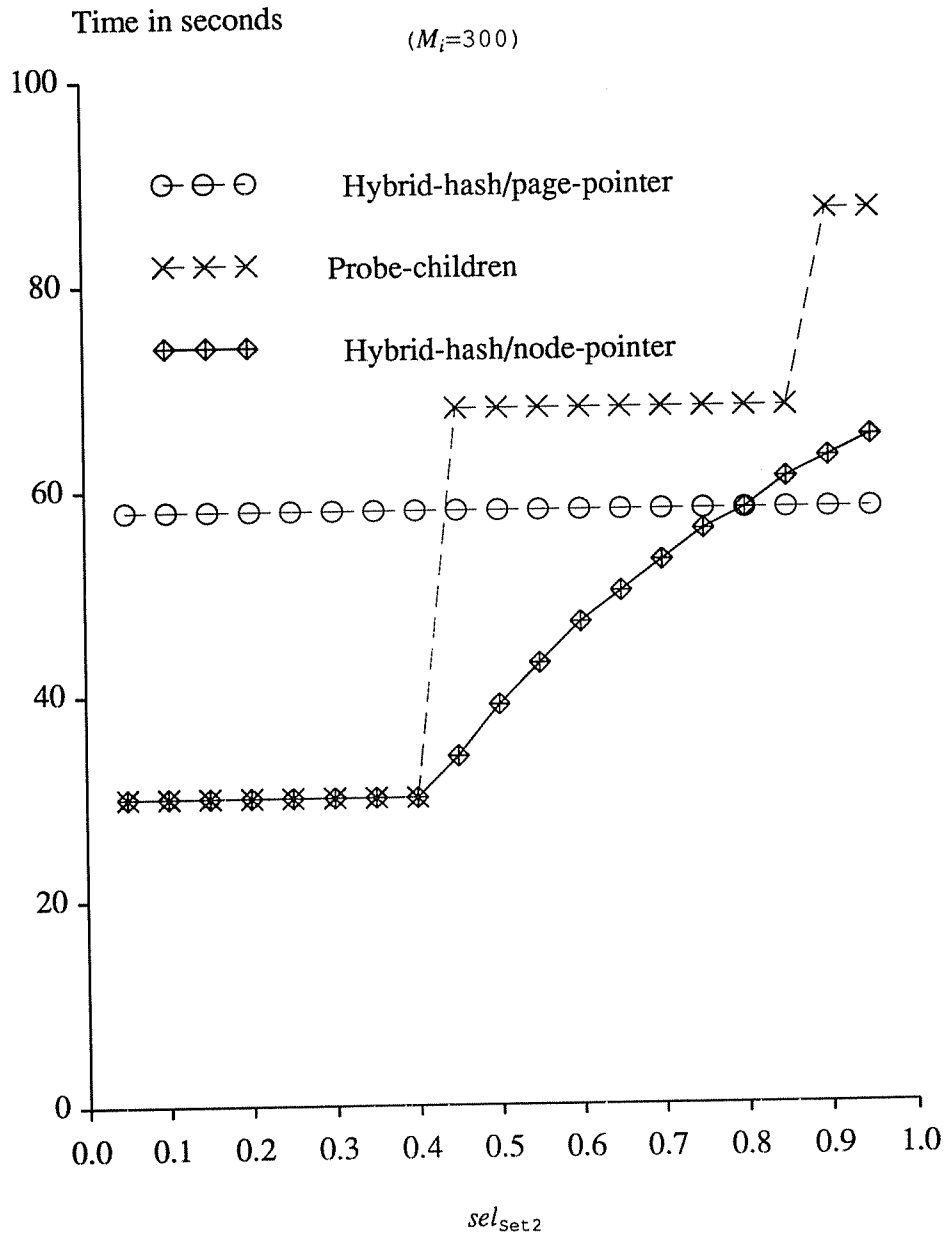

Figure 6.8: Poorly Clustered Database

load-child algorithms require much less space than those for the **load-parent** algorithms (i.e. **Hybrid-hash/page-pointer** and **Hash-loops**). However, the **load-parent** algorithms outperform the **load-child** algorithms for very large memory sizes, since the **load-parent** algorithms then read both `Set1` and `Set2` once. The **load-child** algorithms read `Set1` at least twice: once to compute the `Set2` extent and once to do the join. **Hash-loops** reaches optimal performance with slightly less memory than **Hybrid-hash/page-pointer** because **Hash-loops**'s `Set1`-tuples contain an average of 1.15 pointers each, while those for **Hybrid-hash** contain exactly one. Thus, **Hybrid-**

hash/page-pointer produces 1.15 times as many `Set1`-tuples and, hence, requires more space for its `Set1`-tuples.

Figure 6.8 demonstrates that the **load-child** algorithms work well if there is a fairly restrictive predicate on the inner set. The more restrictive the predicate, the better they perform because they must reread `Set1`-tuples less frequently. The **load-parent** algorithms gain no benefit from a predicate on `Set2` because they cannot apply the predicate until the `Set2` page has already been read. Although its performance does not vary with $sel_{\texttt{Set2}}$, **Hybrid-hash/page-pointer** was included for comparison purposes.

6.3.7.2. Well Clustered Database

The data in the first algorithm comparison had poor clustering of references, so the full potential benefits of the **low-replication** algorithms (i.e. **Probe-children** and **Hash-loops**) were not seen. To illustrate the effects of good reference clustering, consider a database identical to the last one except that ar_i was 2.65 (the number that would be expected if each object at node$_i$ referenced between one and four other nodes). We compare the four algorithms across a range of memory allocations in Figure 6.9. The improved reference clustering does not affect the performance of either **Hybrid-hash** algorithm relative to Figure 6.7—they do the same amount of work because each node receives the same number of `Set1`-tuples. However, the performance of the **low-replication** algorithms improves dramatically because each node now receives 22,944 `Set1`-tuples, (with an average of 2.65 pointers each) instead of 52,870 tuples (with an average of 1.15 pointers each). With good clustering, **Hash-loops** reaches optimal performance long before **Hybrid-hash/page-pointer** because it has far fewer replicated `Set1`-tuples to process. Also, good clustering makes **Probe-children** very competitive with **Hybrid-hash/node-pointer**—as opposed to the situation in Figure 6.7 where **Probe-children** was the clear loser until both **load-child** algorithms reached optimal performance.

Figure 6.10 compares the **load-child** algorithms in the well-clustered database where the memory size is fixed, but the selectivity of the predicate on `Set2` is varied. The performance of the **Hybrid-hash** algorithms is again the same in Figures 6.8 and 6.10, because the same number of replicas are received at each node whether the reference clustering is good or bad. **Probe-children** receives fewer so its performance improves. By avoiding replication, its performance can exceed that of the **Hybrid-hash** algorithms.

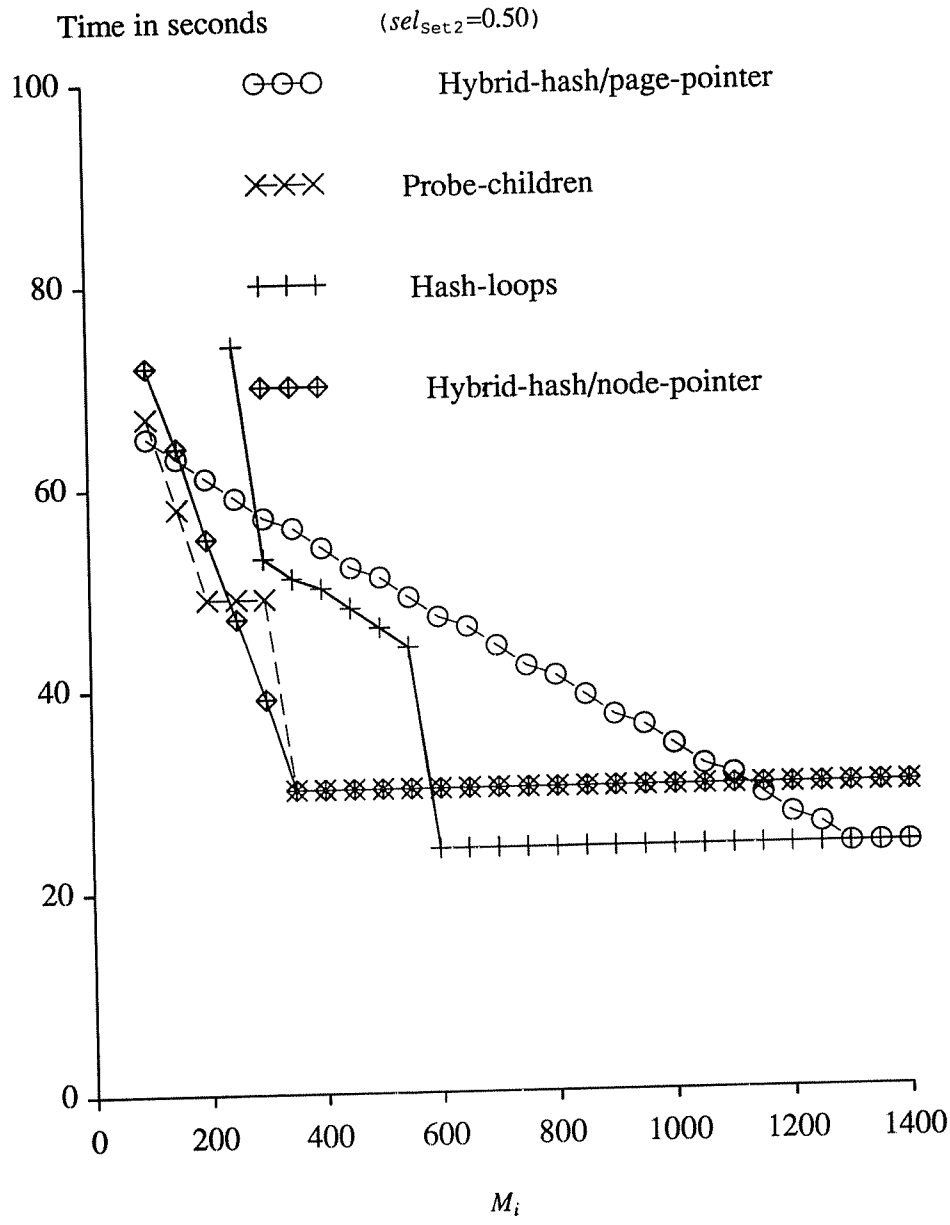

Figure 6.9: Well Clustered Database

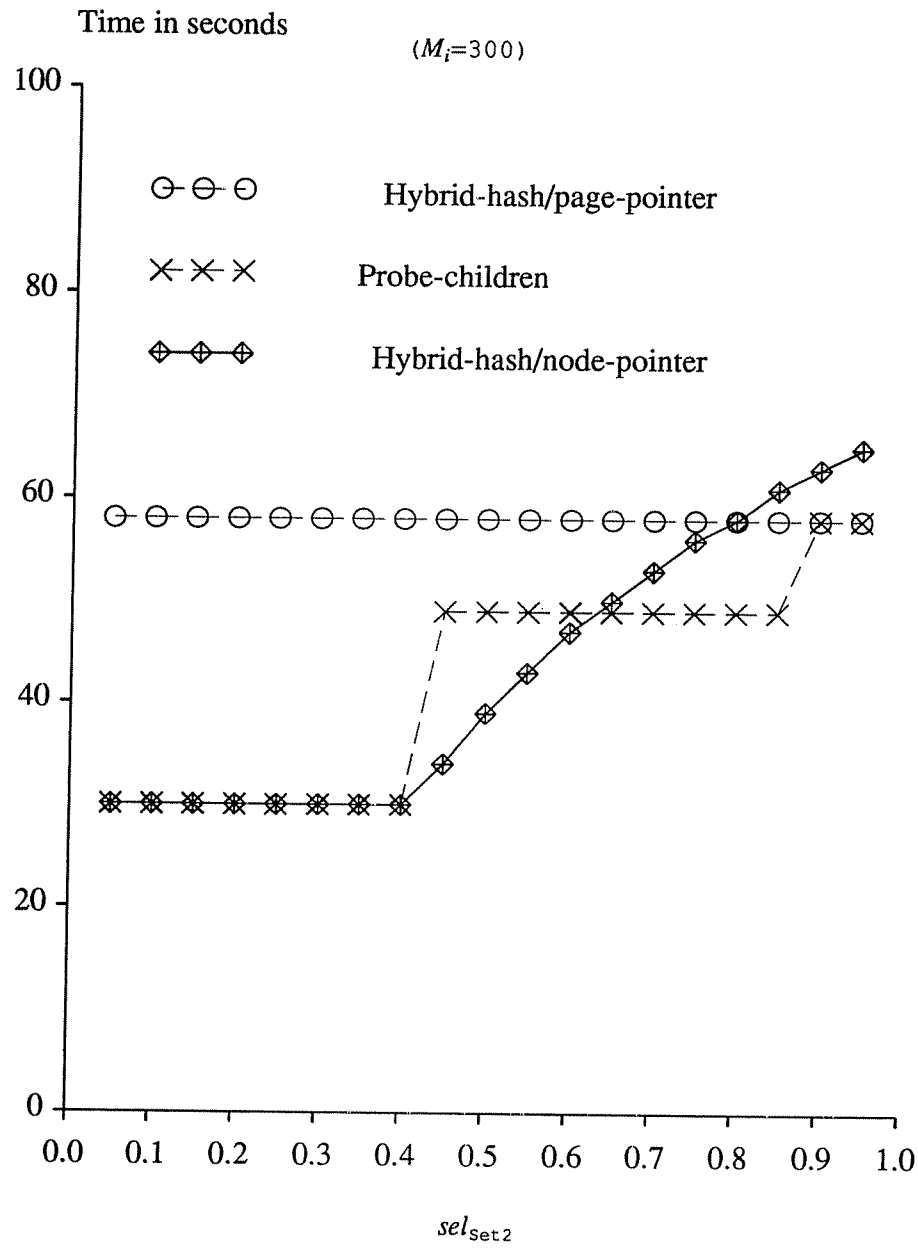

Figure 6.10: Well Clustered Database

6.3.7.3. Database with Tuple Placement Skew

In our next algorithm comparison, we considered a well-clustered database with tuple-placement skew [WALT91]. Since the performance of the whole query is determined by the slowest node, we use 31 evenly balanced nodes, each of which has 29,488 `Set2` objects, and one node with 58,672 `Set2` objects. `Set2` has the same number of elements as in the past comparisons—it is just differently distributed. The $a_i r$ for the most heavily loaded node was 2.65 as in the last example. The algorithms are compared across a range of memory allocations in

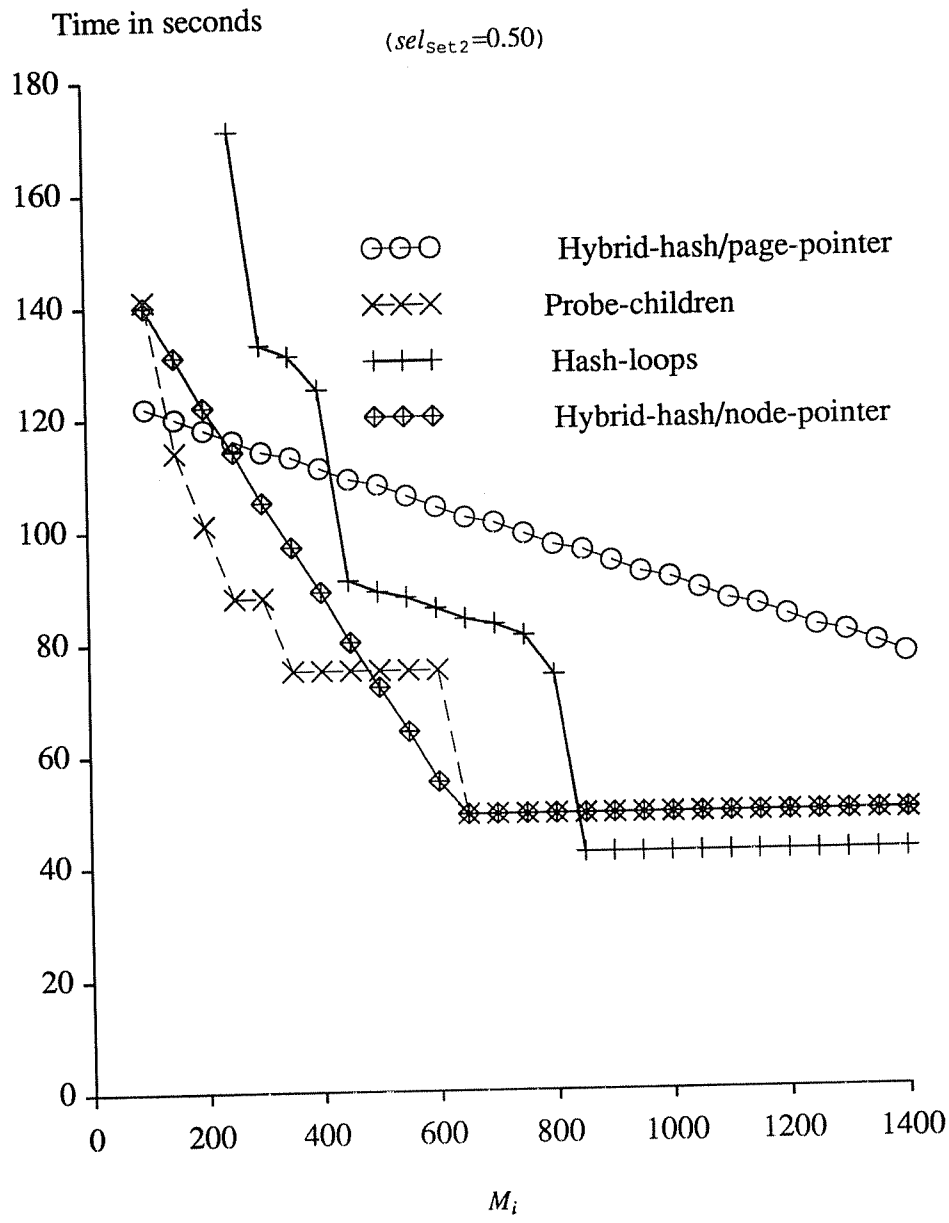

Figure 6.11: Database with Tuple Placement Skew

Figure 6.11. First, we note that the larger `Set2` is, the bigger the payoff of minimizing the replication of `Set1`-tuples, a point that is orthogonal to skew. If tuple-placement-skew on the inner set is significant, using the pointers is too expensive unless the join algorithm's hash table data can fit in a single memory-resident table. Otherwise, it will be better to replicate each `Set1` object once per child pointer, repartition both `Set1` and `Set2` (tagging each `Set2`-tuple with its oid), and use a standard parallel **Hybrid-hash** algorithm [GERB86, SCHN89, DEWI92a] with the oids as join attributes—in which case we would expect performance similar to **Hybrid-hash/node-pointer**

in Figure 6.9 after shifting it up about 19 seconds everywhere to account for having approximately twice as much to read in order to partition `Set2` at the node with tuple-placement-skew. A hash function that ignores the node but uses the page and slot identifier from the pointer should produce fairly uniformly sized partitions. Alternatively, a skew resistant join technique [KITS90, WOLF90, HUA91, WALT91, DEWI92b] might be used after producing `Set1`-tuples. Note that the **Find-children** algorithm must be used to allow either of these techniques if `Set2` is not an explicit extent.

6.3.7.4. Speedup and Scaleup

Next, we compared speedups for the algorithms; we varied the number of nodes, but kept the same number of `Set1` and `Set2` objects as in previous examples. The objects are uniformly distributed across n nodes, where n is varied from 16 to 88. The references are well clustered (ar_i=2.65). Figure 6.12 compares the algorithms' performance. The **load-child** (i.e. **Hybrid-hash/node-pointer** and **Probe-children**) algorithms make relatively modest performance improvements once there are more than 40 nodes. With 40 nodes, their `Set2` hash tables will fit in main memory. **Hash-loops'** performance is poor until most of the `Set1`-tuples at each node will fit into the hash table. Since **Hybrid-hash/page-pointer** has 2.65 times as many tuples to put into its hash table as **Hash-loops**, **Hash-loops** is eventually able to provide better performance. It provides the best performance of any of the algorithms from n=84 on, since then all of its `Set1`-tuples will fit in a hash table. **Hybrid-hash/page-pointer** continues to have the worst performance all the way to the point where adding more processors actually degrades performance (past n=244, which is not on the graph)[12].

If the speedup curves are displayed in typical $\frac{small\_system\_elapsed\_time}{big\_system\_elapsed\_time}$, all the algorithms display super-linear speedup over part of the range if the small system is one with fewer than n=40 nodes, because having one fewer `Set2` object at node$_i$ means that k=10 fewer `Set1`-tuples are sent to node$_i$ ($\frac{10}{2.65}$ fewer for the **low-replication** algorithms). Since the **load-child** algorithms' hash tables fit in main memory when n=40, if n=40 is used as the small system, they have linear speedup beyond that point to at least 164 nodes. The same is true for **Hash-loops** if n=84 is used as the small system.

[12]Speedup performance eventually degrades as more nodes are added because adding a new node requires taking one page away from step (1)'s hash table. Eventually, losing this page hurts performance more than having less data to process helps performance.

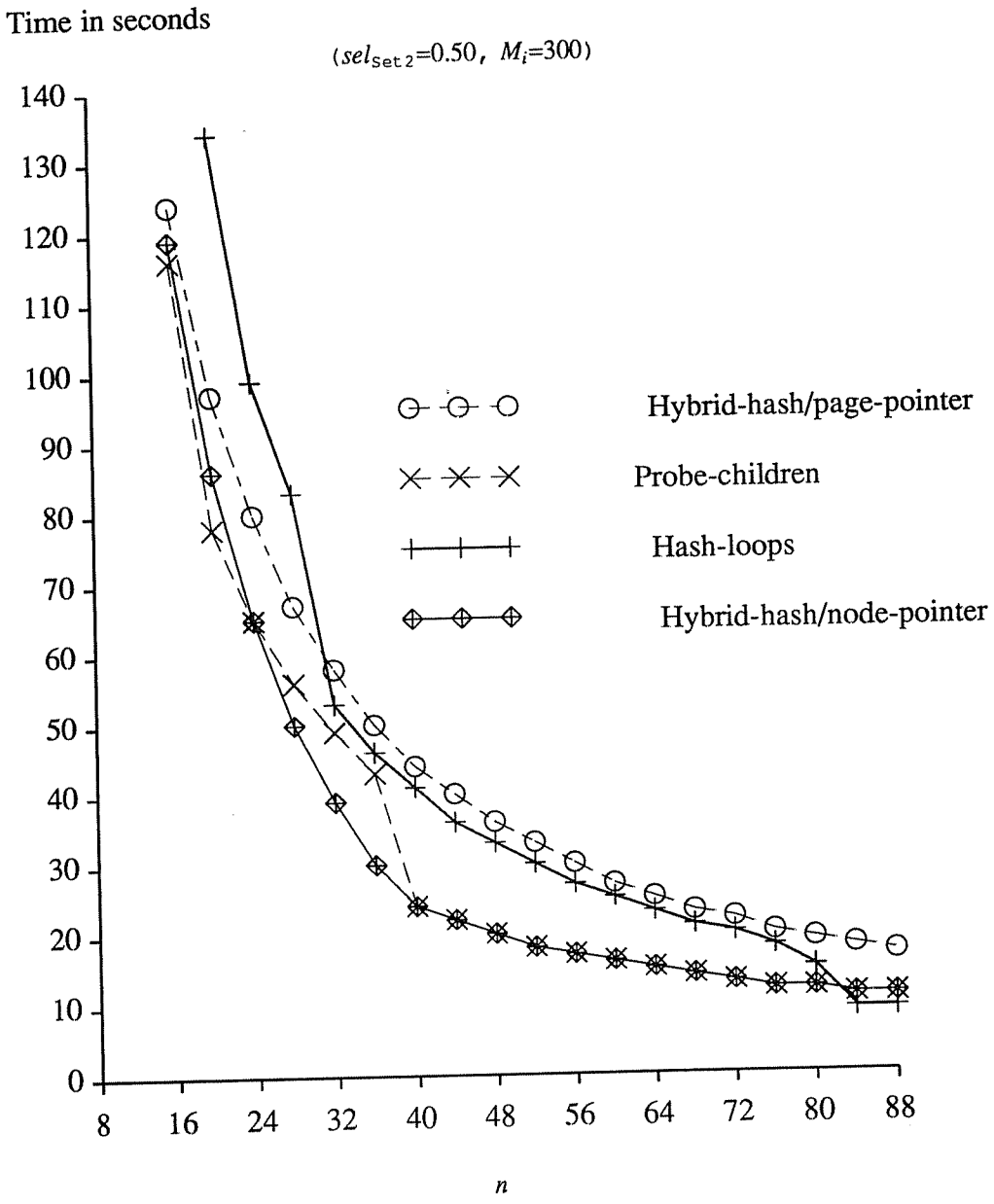

Figure 6.12: Speedup for a Well Clustered Database

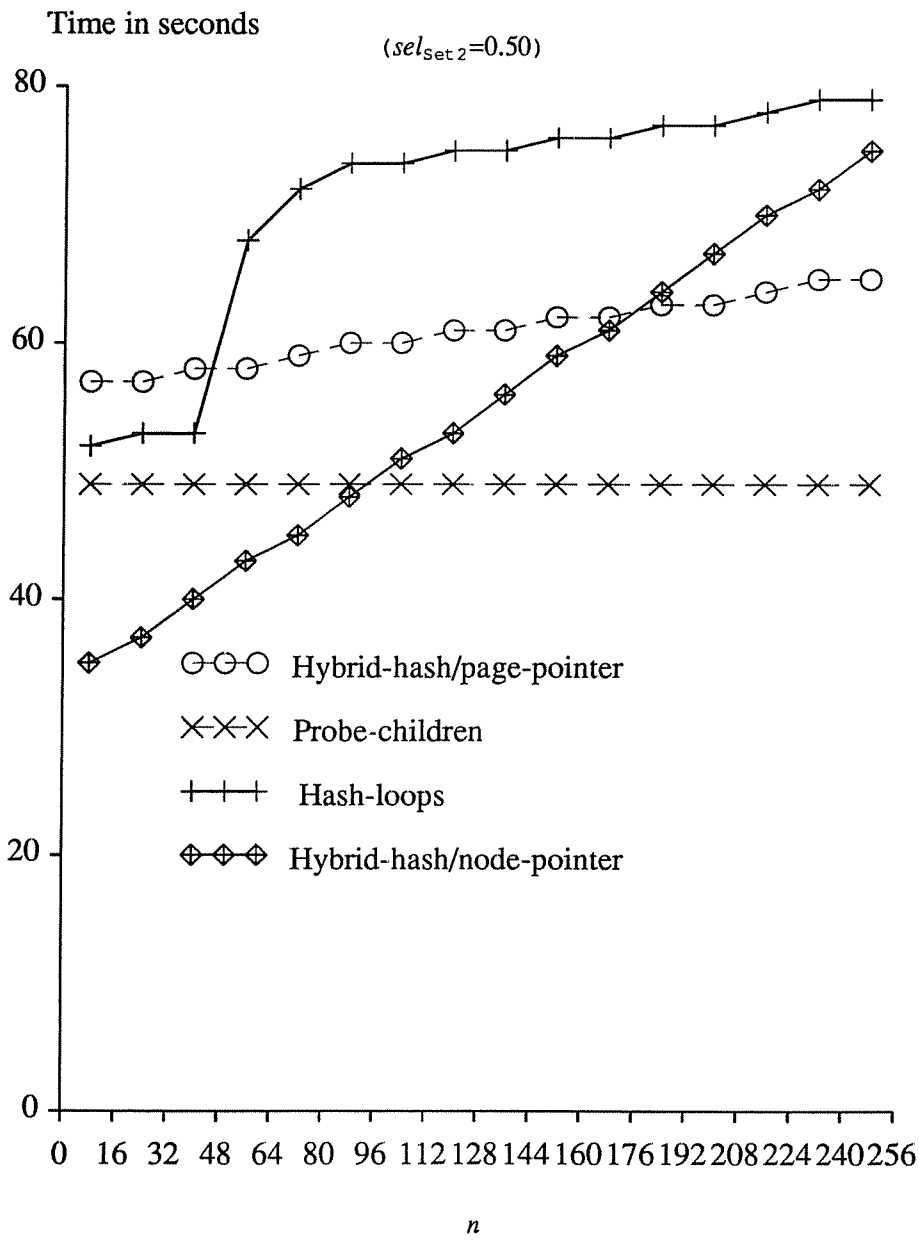

Figure 6.13: Scaleup for a Well Clustered Database

A scaleup algorithm comparison was also run where sel_{Set2}=0.50, $|Set1_i|$=6080, $|Set2_i|$=30400, and M_i=300 $\forall i$ $8 \le i \le 248$, as was the case in several previous comparisons. As seen in Figure 6.13, all of the algorithms except **Hybrid-hash/node-pointer** displayed near-linear scaleup over the range n=8 to 44. After that, the execution time of **Hash-loops** increased rapidly. Adding a new node requires taking one page from the hash table during step (1) of **Hash-loops**, and, eventually, this leads to degrading scaleup performance. Reducing the size of the initial hash table produced smaller performance degradation for the **Hybrid-hash/page-pointer** algorithm, as **Hash-loops**

is much more sensitive to the amount of available memory than it is. The **Hybrid-hash/node-pointer** curve had a slope of about 0.16 (the curve for perfect scaleup has a slope of zero) across the range. It initially had the best performance but its performance became worse than **Probe-children** by n=100 and worse than **Hybrid-hash/page-pointer** by n=184. Initially, because most of the $\texttt{Set2}_i$-tuples fit in main memory at each node_i, its work was roughly a single read of $\texttt{Set2}_i$ and $\texttt{Set1}_i$. However, as adding nodes took more and more pages from the initial hash table, **Hybrid-hash/node-pointer** had to write and read most of the $\texttt{Set2}_i$- and `Set1`-tuples. Since **Probe-children** and **Hybrid-hash/page-pointer** read $\texttt{Set2}_i$ only once, eventually they achieve better performance than **Hybrid-hash/node-pointer**.

6.3.7.5. Summary of Algorithm Comparisons

This section demonstrated that using pointer-based join algorithms can be undesirable if there is tuple-placement-skew on the inner set. If data is relatively uniformly distributed across nodes, however, such algorithms can be desirable; this is because standard **Hybrid-hash**'s performance on replicated `Set1` objects will be roughly comparable to **Hybrid-hash/node-pointer** in this case—good, but not necessarily the best. This section also demonstrated that algorithms that avoid replication can produce significant performance advantages. **Hash-loops** looks much more attractive in an environment where using **Hybrid-hash** requires replication than it did in [SHEK90] provided that most of the `Set1` objects reference `Set2` objects on a small number of nodes[13] and that data is relatively uniformly distributed across nodes. [SHEK90] only examined the performance of algorithms that have sets of pointers from parents-to-children when there was a corresponding child-to-parent pointer; this gave more options and made **Hash-loops** look less attractive. However, in an OODBS, child-to-parent pointers frequently do not exist, and each child may potentially have many parents. Thus, even in a centralized system, replication may be required in order to use **Hybrid-hash** algorithms—making **Hash-loops** a better choice more often than it was in [SHEK90]. We also showed that using **Find-children** and a **load-child** algorithm can be a clear winner at moderate memory sizes. In the presence of tuple-placement-skew, **Find-children** can be indispensable to improving performance because it allows `Set2` to be redistributed across the nodes.

---

[13]It is also more attractive if each set-valued attribute contains a "large" number of oids relative to n. For instance, for n=32, if each set-valued attribute contains 32 oids, a_i^r is expected to be 1.57 even if no clustering was attempted and references are randomly distributed across the n nodes.

The four algorithms we considered must sometimes partition their join stream. We considered a parallel version of [SHEK90]'s pointer-based nested-loops algorithm that had each `Set1` node concurrently request `Set2` pages from other nodes, since we wanted a parallel algorithm that did not require partitioning. We found that this offered only modest performance improvements, and only in the case where the selection predicate on `Set1` was very restrictive. Thus, we concluded that the algorithm was not worth using. However, requiring partitioning does produce a non-uniform join stream since the whole join must be computed before any result tuples can be used. If a more uniform stream (where tuples are produced throughout the join process and not just at the end) is desired, a centralized pointer-based nested-loops algorithm may be a good choice.

6.4. SUMMARY

This chapter began by considering boundary cases cases on the amount of parallelism that can be extracted from a program. One example showed how the **Hybrid-hash** join algorithm can be modified to execute program statements rather than produce result tuples. The conditions under which this modified **Hybrid-hash** algorithm can be used to execute a join loop while maintaining program semantics were discussed. If these conditions are met, a join loop can be parallelized to the same extent as a relational join. We then considered an example of a set iteration that must be performed at a central site to maintain program semantics.

Having seen the range of parallelism opportunities, we then briefly considered how the transformations of Chapter 3 can be used to enhance parallelism opportunities. We showed how a general group-by loop could be parallelized, and we then showed why applying (T5) to replace the general group-by loop with three simpler loops might be advantageous.

In addition to exploring parallelization via transformations, we described and analyzed four parallel join algorithms for set-valued attributes. We also presented the **Find-children** algorithm which can be used to compute an implicit extent of objects referenced. If an explicit extent does not exist, using this algorithm gives the system much more flexibility in how it evaluates the join.

Experiments performed using our analysis demonstrated that the **Hash-loop** and **Probe-children** join algorithms can be very competitive with parallel pointer-based **Hybrid-hash** join algorithms. These pointer-based join algorithms show great promise for parallelizing a DBPL. Since some of the pointer-based joins were originally proposed for use by centralized relational database systems with referential integrity support [SHEK90], these algorithms should also be useful for such relational systems.

CHAPTER 7

SUMMARY

7.1. CONCLUSIONS

Database Programming Languages have been proposed as a solution to the *impedance mismatch* problem resulting from the limited expressive power of relational query languages. However, unless DBPLs can provide the same level of performance as relational database systems for large databases, they will not succeed in replacing relational database systems except in niche markets (like CAD, for projects whose working set of the database fits in main memory).

Achieving comparable performance is made more difficult by the imperative nature of the set iteration constructs of DBPLS such as PASCAL/R, O_2, E, and O++. Nesting set iterators is a natural way to express a join—particularly for programmers without database experience, who are used to programming in an imperative language (one of the main target markets for DBPLs). The sequential semantics of the language specify that such a join must be evaluated centrally by a tuple-at-a-time nested-loops join algorithm unless program analysis demonstrates that some other (perhaps parallel) join algorithm will maintain proper program semantics. Thus, maintaining program semantics while improving performance requires careful dataflow analysis—much like the vectorization of sequential FORTRAN programs. The query languages of relational database systems, on the other hand, are non-procedural. This simplifies optimization enormously since no dataflow analysis is needed.

This thesis used dataflow analysis to identify nested set iterators that can be optimized like relational joins. It also proposed program transformations that replace a nested set iterator with several simpler set iterators that may be more amenable to optimization than the original iterator. These transformations together with transformations for relational joins can be applied to a nested set iterator to produce a (potentially large) number of equivalent plans. Some of them will be better—and others will be worse—than the original set iterator. Choosing the wrong plan for evaluating a computation over the database may be very costly, so cost functions must be used to search a space of plans to find an "optimal" plan for computing the result. This thesis combined the dataflow analysis and program transformation techniques from the programming language community (for correctness) with the search techniques of the database community (for efficiency). The thesis was organized as follows.

In Chapter 3, we considered six program transformations that can be used to rewrite nested set iterators into a more efficient form. Cost formulas and analyses were used to demonstrate that these transformations can produce significant performance improvements.

Chapter 3 made extensive use of the concept of self-commutativity. The concept was used to identify nested iterators that can be optimized like relational joins. Chapter 3 defined the concept and gave examples, but it did not characterize the class of self-commutative statements. Chapter 4 characterized a subclass of self-commutative statements and proved the correctness of the characterization.

Chapter 5 described how the transformations of Chapter 3 can be formulated as tree rewrites. This allows standard database style transformation-based optimization to be performed. An optimizer based on this technique was built using the EXODUS Optimizer Generator and added to the AT&T Bell Labs *O++* compiler. The resulting optimizing compiler was used in experiments that demonstrated that these transformations can produce significant performance improvements in a centralized system.

Chapter 6 considered how these transformations can be used to enhance parallelism opportunities. Examples of how the parallel **Hybrid-hash** join algorithm can be modified to execute program statements rather than produce result tuples were given. The conditions under which this modified **Hybrid-hash** algorithm can be used to execute a nested set iterator while maintaining program semantics were discussed. If these conditions are met, a join loop can be parallelized to the same extent as a traditional relational join. We then described, analyzed, and evaluated several parallel pointer-based join algorithms, one of them new, all of them previously unanalyzed. They show great promise for parallelizing DBPLs, and they may also be useful for (extended) relational database systems.

7.2. FUTURE RESEARCH DIRECTIONS

Future work derived from this thesis includes finding new transformations, particularly transformations that can combine several loops that appear sequentially in the program text into a single large loop (in some ways finding an inverse of transformation (T4)—closely related to multi-query optimization [SELL88]). We currently do combine some loops produced by the optimizer while transforming a single group-by loop. However, we would like to apply this technique more widely.

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 an iteration through the `Dept` extent in alphabetical order of the `Dept` names.)

We would also like to formalize our tree-based representation of rewrites described Chapter 5. It is currently an implementation technique; we would like to produce an algebra.

Incorporating object sharing dataflow analysis of the sort described in [LARU89] is also a priority. Our dependence analysis is currently very conservative, and we would like to improve it.

Applying the techniques developed here to nested SQL cursors embedded in COBOL or *C* would also be interesting. This is a common programming paradigm—because some users don't trust the optimizer, while others need functionality like outerjoins that their system does not provide [GRAY92]. These applications may eventually need to be run on a parallel system, and to gain the benefits of parallelism, these programs must be rewritten to remove the nested cursors. This is a time-consuming task that is likely to add programming errors. Our techniques could be used to at least partially automate the process. Thus, they may be useful to enable a smooth transition to a new parallel database system.

We wish to develop techniques to parallelize a wider class of programs. In addition, we are interested in developing new parallel algorithms for processing the bulk-data structures of an OODBS. Finally, we would like to actually build a parallel version of a DBPL (probably *O++*) that employs our transformations.

REFERENCES

[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.

[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 Int. Conf. on DOOD (Deductive and Object-Oriented Databases)*, Japan, December, 1989.

[BATE90] Samuel Bates, private communication.

[BEER90] C. Beeri and Y. Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. *Proc. 1990 Int. Conf. Database Theory*, December 1990.

[BUTT91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone Object Database Management System. *CACM 34,10* (October 1991), 64-77.

[CARE86] Michael Carey, David DeWitt, Joel Richardson, and Eugene Shekita. Object and File Management in the EXODUS Extensible Database System. *Proc. 1986 Conf. Very Large Databases*, August 1986.

[CARE90] Michael Carey, Eugene Shekita, George Lapis, Bruce Lindsay, and John McPherson. An Incremental Join Attachment for Starburst. *Proc. 1990 Conf. Very Large Databases*, August 1990.

[DAYA87] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Process Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. *Proc. 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.

[DEUX91] O. Deux et al. The O_2 System. *CACM 34,10* (October 1991), 34-48.

[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.

[DEWI91] David DeWitt, Jeffrey Naughton, and Donovan Schneider. A comparison of non-equijoin algorithms. *Proc. 1991 Conf. Very Large Databases*, August 1991.

[DEWI92a] David DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM 35,6* (June 1992), 85-98.

[DEWI92b] David DeWitt, Jeffrey Naughton, Donovan Schneider, and S. Seshadri. Practical Skew Handling Algorithms For Parallel Joins. *Proc. 1992 Conf. Very Large Databases*, August 1992, to appear.

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

[GERB86] Robert Gerber. Ph.D Thesis. Dataflow Query Processing using Multiprocessor Hash-partitioned Algorithms. University of Wisconsin (1986).

[GHAN90] Shahram Ghandeharizadeh. Ph.D Thesis. Physical Database Design in Multiprocessor Database Systems. University of Wisconsin (1990).

[GRAY92] Jim Gray, private communication.

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

[GRAE90] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. *Proc. 1990 SIGMOD*, May 1990.

[HART88] Brian Hart, Scott Danforth, and Patrick Valduriez. Parallelizing a Database Programming Language. *1988 Int. Symposium on Databases in Parallel and Dist. Syst.*, December 1988.

[HART89] Brian Hart, Patrick Valduriez, and Scott Danforth. Parallelizing FAD using Compile-time Analysis Techniques. *Data Engineering 12,1* (March 1989), 9-15.

[HUA91] Kien Hua and Chiang Lee. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. *Proc. 1991 Conf. Very Large Databases*, September 1991.

[JARK84] Matthias Jarke and Jurgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys 16,2* (June 1984), 111-152.

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

[KHOS86] Setrag Khoshafian and George Copeland. Object Identify. *Proc. 1986 OOPSLA*, September 1986.

[KIM80] Won Kim. A New Way to Compute the Product and Join of Relations. *Proc. 1980 SIGMOD*, May 1980.

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

[KITS90] Masaru Kitsuregawa and Yasushi Ogawa. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC). *Proc. 1990 Conf. Very Large Databases*, August 1990.

[KORT91] Henry Korth and Abraham Silberschatz. *Database System Concepts (2nd edition).* McGraw-Hill, New York, 1991.

[LAMB91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore Database System. *CACM 34,10* (October 1991), 50-63.

[LARU89] James Larus. Ph.D Thesis. Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors. University of California (1989).

[LIVN87] M. Livny, S. Khoshafian, and H. Boral. Multi-Disk Management Algorithms. *Proc. 1987 SIGMETRICS*, May 1987.

[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. *Proc. 3rd Int. Workshop on Database Programming Languages*, August 1991.

[LIEU92] Daniel Lieuwen and David DeWitt, A Transformation-based Approach to Optimizing Loops in Database Programming Languages. *Proc. 1992 SIGMOD*, June 1992.

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

[MACK89] Lothar Mackert and Guy Lohman. Index Scans Using a Finite LRU Buffer: A Validated I/O Model. *ACM Trans. Database Syst. 14,3* (September 1989), 401-424.

[MEHT91] Manish Mehta and David DeWitt. Pointer Join Techniques for Parallel Database Machines, manuscript.

[MUMI90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, Raghu Ramakrishnan. Magic is Relevant. *Proc. 1990 SIGMOD*, June 1990.

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

[OMIE89] Edward Omiecinski and Eileen Tien Lin. Hash-Based and Index-Based Join Algorithms for Cube and Ring Connected Multicomputers. *IEEE Trans. Knowledge and Data Engineering 1,3* (September 1989), 329-343.

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

[RIES78] D. Ries and R. Epstein. Evaluation of Distribution Criteria for Distributed Database Systems. UCB/ERL Technical Report M78/22, UC Berkeley, May 1978.

[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.

[RICH92] Joel Richardson, Michael Carey, and Daniel Schuh. The Design of the E Programming Language. *ACM Trans. Programming Languages and Syst.* (1992), to appear.

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

[SCHN89] Donovan A. Schneider and David J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. *Proc. 1989 SIGMOD*, June 1989.

[SELI79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. *Proc. 1979 SIGMOD*, May-June 1979.

[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.

[SHAW89] Gail Shaw and Stanley Zdonik. An Object-Oriented Query Algebra. *Data Engineering 12,3* (September 1989), 29-36.

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

[SHEK91] Eugene Shekita. Ph.D Thesis. High-Performance Implementation Techniques for Next-Generation Database Systems. University of Wisconsin (1991).

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

[VALD87] Patrick Valduriez. Join Indices. *ACM Trans. Database Syst. 12,2* (June 1987), 218-246.

[VAND91] Scott Vandenberg and David DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. *Proc. 1991 SIGMOD*, May 1991.

[VOSS91] Gottfried Vossen. *Data Models, Database Languages and Database Management Systems*. Addison Wesley, New York, 1991.

[WALT91] Christopher Walton, Alfred Dale, and Roy Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. *Proc. 1991 Conf. Very Large Databases*, September 1991.

[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.

[WOLF90] Joel Wolf. Daniel Dias, Philip Yu, and John Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. IBM T. J. Watson Research Center Technical Report RC 15510, 1990.

[YAO77] S. B. Yao. Approximating Block Accesses in Database Organizations. *CACM 20,4* (April 1977), 260-261.

[ZDON90] Stanley Zdonik and David Maier. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, California, 1990.

[ZWILL92] Michael Zwilling, private communication.