Translating Aggregate Queries into Iterative Programs

Johann Christoph Freytag[1]
IBM Almaden Research Center, San Jose, CA 95120
Nathan Goodman
Kendall Square Research Corp., Cambridge, MA 02139

Abstract

Over the last decade, many techniques for optimizing relational queries have been developed. However, the optimization of queries with aggregation has received little attention.

This paper investigates possible improvements for aggregate queries on groups of tuples. We suggest the use of program transformation methods to systematically generate efficient iterative programs for their evaluation. Two transformation steps successively translate a program, which sorts the relation before applying the aggregate function, into an iterative program which performs the aggregation while sorting, thus using less time and space than needed for the execution of the initial program.

1. Introduction

Many database management systems (DBMSs), such as SYSTEM R and INGRES, permit the computation of counts, sums, averages and other aggregate quantities [ASTR76, STONE76]. Aggregation is usually performed over a set of tuples yielding a singleton value. We can also partition, or group, tuples of a relation into subsets according to common values of one or more attributes. The aggregation is then performed over each of the subsets yielding a *set of singleton values*. For example, consider the SQL query:

```
SELECT SUM(Salary) FROM EMP
GROUP BY Department
```

on Relation EMP (EMP#, Salary, Department). This query computes the sum of salaries for each department in the EMP relation. As the number of departments is not known in advance, one possible evaluation strategy is to *sort* the relation EMP on department values before the salary sum for each department is computed by scanning the sorted relation once.

This processing strategy is not optimal. Klug notices that in many cases the aggregate computation - or at least part of it - may be performed *while sorting* [KLUG82]. For instance, in the above example, whenever two tuples are compared during the sort *and* they agree in their department values, we can merge them into a *new tuple* by adding their salaries. The sort then continues with the newly generated tuple. Applying this change throughout the sort, the final result is a relation, sorted on the department values, with *exactly one tuple* for each department recording its salary. An additional scan of the relation to perform aggregation becomes unnecessary. An additional gain in execution speed results from a decreasing number of tuples during the sort, thus requiring less time for sorting and less storage space. Klug suggests a specially designed sort function which implements the combination of sorting and aggregation [KLUG82].

In this paper we derive the suggested improvements for aggregate functions by methods of program transformation [BACK78, DARL76, BURS77]. We generate the efficient programs, as described above, from initial specifications which independently define the aggregate function and the sort algorithm.

The transformation consists of two phases. The first phase, which we call the *horizontal transformation phase*, transforms the initial program specifications for sorting and aggregation into the desired optimal program form. Horizontal transformation manipulates high-level program specifications which are based on Backus' FP language [BACK78]. To guarantee fast execution, the second phase, which we call the *vertical transformation phase*, translates the high level specification into an iterative program in a PASCAL-like notation.

To make the discussion of the transformation more concrete, we choose one sort algorithm and one aggregate function. To implement sorting we use the *sort-merge* algorithm [KNUT73]. We demonstrate all transformation steps using the aggregate function *sum_by* which performs the summation of values on groups of tuples. Later we show how the proposed transformation generalizes to other aggregate functions as well.

The paper is organized as follows. In the next section we provide some more motivation for using methods of program transformation to generate efficient programs for the evaluation of aggregate functions. In Section 3 we introduce the notation for describing the sort and the aggregate function before Section 4 develops the transformation steps to generate the improved program. In Section 5 we generalize the transformation of Section 4 to other aggregate functions by introducing a uniform definition which is better suited for formal manipulation.

[1] This research was done while the author was a student at Harvard University, Cambridge, MA 02138. The work was supported by the Office of Naval Research under grant ONR-N00014-83-K-0770.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

2. Motivation

Query languages for relational DBMSs, such as QUEL [STONE76] or SQL [ASTR76], permit the user to express database requests in a clear and understandable form that describes properties of the requested result without considering aspects of an efficient execution. Therefore it is the DBMS's responsibility to derive an evaluation strategy to compute the query efficiently. The component of the DBMS which decides on the strategy is called the *query optimizer*. Based on the information about the internal representation of the data accessed and the available evaluation strategies, the query optimizer produces a *query evaluation plan (QEP)*. For example, the existence of indices or information about the physical order of elements in the relation influences the optimizer's choice of either an index-join, a merge-join, or a nested-loop join as the optimal evaluation strategy for a join operator. Jarke and Koch give a comprehensive overview on various optimization techniques for relational queries [JARK84].

After its generation by the query optimizer, the QEP has to be executed against the database to compute the requested result. To perform this task, many DBMSs face a translation problem since the QEP most often uses operations on *sets of tuples*, while the execution system performs operations on *tuples* to ensure an efficient execution [ASTR76, STONE76]. This gap between the generated QEP and the query execution system introduces the problem of mapping QEPs into sequences of operations for the execution system. We call this problem the *query translation problem* [FREY85a, FREY85b].

In many ways, both translation steps, i.e. optimization and query translation into operations for a specific execution system, resemble a specialized *program transformation problem*. Generally speaking, program transformation promises to provide a comprehensive solution to the problem of producing programs which try to solve several incompatible goals simultaneously: On the one hand, programs should be correct and clearly structured, thus allowing easy modification. On the other hand, one expects them to be executed efficiently. Using languages like C or PASCAL for implementation of programs, one is immediately forced to consider aspects of efficiency that are often unrelated to their correctness, natural structure, and clarity.

For this reason, the transformational approach tries to separate these two concerns by dividing the programming task into two steps: The first step concentrates on producing programs that are written as clearly and understandably as possible without considering efficiency issues. If initially the question of efficiency is completely ignored, the resulting program might be written very comprehensibly, but might be highly inefficient or even unexecutable. The second step then successively transforms programs into more efficient ones - possibly based on additional information - using methods which preserve the meaning of the original program.

In our opinion, these intentions exactly guided the design of query languages for relational DBMSs where queries are expressed in a data independent manner. We therefore argue that methods which have been developed in the area of program transformation may prove useful for the translation of relational queries. In [FREY85b] we describe the transformation of algebra-based query specifications into iterative programs which are executable on a tuple-oriented query execution system. The transformation involves the rule-based manipulation of recursively defined programs and the replacement of recursion by iteration both of which were the focus of early research in program transformation [DARL76, BURSS77, COHE80].

In 1978 John Backus introduced the FP language which motivated a new kind of programming [BACK78]. Much of the current research in program transformation has centered around this new approach [BIRD84, BELL84]. Many researchers describe their transformations in terms of *rewriting rules* or *transformation rules* [HUET80]. These provide a uniform description of the intended transformation without considering implementation details.

The latter approach to program transformation motivated us to apply those methods and ideas to the translation of relational queries. The translation of aggregate functions is a nontrivial example to demonstrate several aspects of a rule-based translation. We also argue that the successive translation of the sort function and the aggregate function into a more efficient program has several advantages over a hand-coded program to implement the desired improvement.

First, independently defined functions for sorting and aggregation support the design of a modular, high-level interface for query evaluation. Deriving desirable improvements from the combination of sorting and aggregation relieves the optimizer from considering another operation during optimization, thus simplifying the optimizer's task.

Second, the use of rules allows a simple description of the performed transformation which may easily lead to an experimental implementation of a query translation system. Each rule performs a sound transformation which guarantees that the newly generated program correctly performs the computation initially specified.

Third, the transformation may be effective even if the aggregate function is not known to the system in advance. The suggested transformation could be applied to combine a user provided function with internally defined processing strategies to guarantee a more efficient query execution.

Our goal for the translation of functions *sort__merge* and *sum__by* is twofold. On the one hand we would like to use the FP notation to perform high level transformations. On the other hand we have to guarantee fast execution of user-submitted queries. To accomplish both goals, we introduce two kinds of transformation: *Horizontal transformation and vertical transformation*. During the first transformation we manipulate programs, specified in an FP-like notation, to generate the program which performs aggregation while sorting. The high-level specification simplifies the manipulation of functions and control structures without considering details of their implementation. Vertical transformation then translates the high level program description into an iterative program to guarantee fast execution.

3. The Definition of Sort_Merge and Sum_By

This section defines the functions *sort__merge* and *sum__by*. For the horizontal transformation we describe the function *sort__merge* in an FP-like notation using the three operators *map, tree,* and *L*. Operators *map* and *tree* specify two different control structures. They determine how to access sets of tuples during sorting. During the transformation we create new functions by combining existing ones. We therefore introduce operator *L* to express the combination of functions. Throughout this paper we use *lists* instead of relations to represent sets of tuples. We prefer lists over relations to avoid confusion and to guarantee notational uniformity.

Operator *map* is motivated by the corresponding Lisp operator which denotes a loop-like control structure without specifying its exact implementation. The meaning of the expression (*map f list*) with *f* being a function of arity one is defined as " applying function *f* to each element in the *list.*" For example, consider *f* to be the square function *sqr* and *list* to be the list <1, 5, 3>, then (*map sqr list*) returns the three element list <1, 25, 9> without specifying the order in which we apply function *sqr* to the initial list. Intuitively, operator *map* specifies to access the tuple list by applying function *f* to each tuple individually.

In the above example we introduced a list of integers. However, we may also use lists containing other kinds of elements. For the definition of *sort__merge* the elements of a list can be lists of elements. For example, if *list* consists of two lists <5, 1, 3> and <2, 6, 3>, and if *sort* denotes the sort function, then (*map sort list*) returns a list of sorted lists

<<1, 3, 5>, <2, 3, 6>>

by applying function *sort* to each element, a list of integers, in the original list.

For the FP-like specification of the sort-merge algorithm, we need to introduce another control structure operator, called *tree*, which denotes a "divide and conquer" control structure [BACK78, WILL82]. (*tree f list*) with *f* being a function of arity two divides a list recursively into a left and a right part until the list contains one element. On each level of the recursion *f* is applied to the results returned form the previous level of recursion. For example, let + denote the addition function and let *list* be <1, 5, 7, 4>, then (*tree* + *list*) performs the computation as shown in Figure 1 returning the expected result of 17.

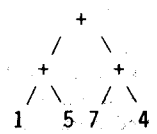

Figure 1: Computation tree for (*tree* + <1, 5, 7, 4>)

Formally, *tree* is defined by

$$(tree\ f\ list) = \begin{cases} list\ , \text{ if the list has one element} \\ (f\ (tree\ f\ (lf\ list))\ (tree\ f\ (rg\ list))) \end{cases}$$

where *lf* and *rg* are functions which divide a list into a left and a right part of (almost) equal length, i.e.

$$(lf < e_1, \ldots, e_n >) = < e_1, \ldots, e_{n/2} >$$

$$(rg < e_1, \ldots, e_n >) = < e_{n/2+1}, \ldots, e_n >$$

The third operator, *L*, is used for the combination of functions during the transformation. Informally, let f_1 and f_2 be two functions of arity *m* and *n*, respectively, then $(L\ f_1\ f_2)$ denotes the combination $(f_1\ (f_2\ x_1 \ldots x_n)\ y_2 \ldots y_m)$ of both functions substituting f_2 for f_1's first parameter.

We use the operators *map* and *tree* to define the *sort-merge* algorithm which performs sorting as follows [KNUT73]. During the first phase a list of tuples is divided into *runs*, i.e. tuples lists of equal length. Each run is sorted before starting the merge-phase. This phase begins with pairs of sorted runs that are merged into one sorted run. Each subsequent phase merges pairs of runs, produced in the previous phase, into a sorted new run which is twice as long as the input runs. Figure 2 shows an example of the merge phase.

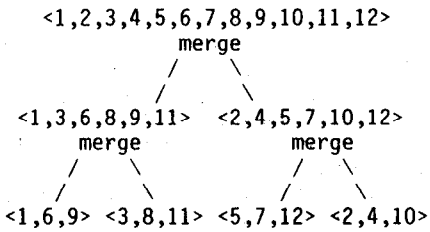

Figure 2: Sort-Merge Computation Tree

The *tree* operator exactly resembles the computational behavior performed during the second phase. According to the explanation we define the merge-sort (*merge__sort$_A$ list*) by the expression

(*tree merge$_A$* (*map sort$_A$* (*convert list*)))

where function *convert* converts a list of elements into runs, i.e. a list of tuple lists. Those runs are subsequently sorted on values of attribute *A* as specified by (*map sort$_A$ list*) before the merge phase begins. Whenever merge is applied to two sorted lists, the result is a sorted list consisting of elements of both inputs. The "tree-like" application of merge ultimately returns one sorted run of elements.

For a complete definition of the merge-sort we need to specify functions *sort, merge*, and *convert*. As function *convert* is not important for the later transformation we do not provide a detailed definition. We define function (*sort$_A$ list*) by another tree expression (*tree merge$_A$ list*) which specifies to recursively split the tuple list into a left and a right part before merging them back again, based on the *A* values of the individual tuples. With the definitions of *merge__sort$_A$* and *sort$_A$* it remains to define function *merge$_A$* which we could implement by a PASCAL-like program. However, for clarity and ease of manipulation we prefer a recursive definition in a Lisp-like notation which uses the following operations on lists[2]:

(*first list*)
returns the first element of the list.

(*rest list*)
returns a new list by removing the first element of the list.

empty
returns the empty list

(*out ele list*)
creates a new list whose first element *ele* is followed by all elements in *list.*

(*empty? list*)
evaluates to true if the list is empty and to false if list is an expression of the form (out ele list).

To define program *merge$_A$* we propose a small functional language in a Lisp-like notation which is well suited for formal manipulation. We shall describe the target language informally. The language is

2 These functions resemble the well known Lisp functions car, cdr, (), cons, and null?, respectively.

based on expressions. An expression is either a variable, a function expression, or a conditional expression. A function expression has the form $(f_1\ t_1 ... t_n)$, where f_1 is a function symbol and t_i, $i = 1, ..., n$ are expressions called actual parameters. A conditional expression has the form $(if\ t_1\ t_2\ t_3)$, where t_1, t_2, t_3 are expressions. If t_1 evaluates to true, then the value of t_2 is the value of the expression, otherwise it is the value of t_3.

We define the function $(merge_A\ l_1\ l_2)$ which merges two lists l_1, l_2, sorted on attribute A, as follows. Let $gr_A?$ and $eq_A?$ be two functions comparing two elements e_1, e_2. If the A value of e_1 is greater (equal) than the A value of e_2, then the functions return true, otherwise they return false:

```
(if (empty? l1) l2
  (if (empty? l2) l1
    (if (grA? (first l1) (first l2))
      (out (first l2) (mergeA l1 (rest l2)))
      (if (eqA? (first l1) (firstl2))
        (out (first l2) (mergeA l1 (rest l2)))
        (out (first l1) (mergeA (rest l1) l2))))))
```

The program for $(merge_A\ l_1\ l_2)$ first checks if one of the lists is empty. If so, either l_1 or l_2 is returned. If both lists have at least one element, those are compared on the values of A. If the A value of the first element in l_2 is less than or equal to the A value of the first element in l_1, we output the first element of l_2 and continue the merge recursively after having removed the first element from l_2. In case the A value of the first element in l_1 is less, that element is returned followed by all elements resulting from the recursive call $(merge_A\ (rest\ l_1)\ l_2)$.

Similarly, we define the aggregate function $(sum\_by_{B,A}\ l_1)$ which computes the sum of the B values for the set of tuples having the same A values. To compute the result correctly, $sum\_by_{B,A}$ expects an input list which is sorted on the A values. Its definition uses the function $(add_B\ e_1\ e_2)$ which adds the B values of e_1 and e_2, creating a new tuple $<b_1 + b_2, a>$ with $e_i = <b_i, a>$. $(sum\_by_{B,A}\ l_1)$ is defined by the expression

```
(if (empty? l1) *empty*
  (sumB,A (first l1) (rest l1)))
```

where $(sum_{B,A}\ ele\ l_1)$ is defined by

```
(if (empty? l1) (out ele l1)
  (if (eqA? ele (first l1))
    (sumB,A (addB ele (first l1)) (rest l1))
    (out ele (sumB,A (first l1) (rest l1)))))
```

If l_1 is not empty, $sum\_by_{B,A}$ calls the two parameter function $sum_{B,A}$. The first parameter serves as an "accumulator" for summing up the B values for the same A value. If a new A value is encountered, the "accumulated" element is returned followed by all elements resulting from the recursive call to function $sum_{B,A}$.

4. The Transfomation of Sort_Merge and Sum_By

Based on the definitions of the previous section, this section describes the transformation of the QEP

$(sum\_by_{B,A}\ (merge\_sort_A\ list))$

into an executable program. The operations, specified by this QEP, are to first sort the *list* based on values of attribute A before applying the aggregate function $sum\_by_{B,A}$ which computes the same of B values for each A value in the list. We carry out the transformation in two phases. First, *horizontal transformation* performs *structural manipulation* using the tree notation for the merge-sort function. We show how the transformation intertwines both functions yielding a program which computes the aggregate function while sorting. Although the tree notation is well suited for the transformational purposes, its recursive nature does not guarantee an efficient execution. For this reason the *vertical transformation* replaces the "tree recursion" by iteration. We present two transformations into iterative programs which differ in their computational behavior. The first program generated maintains a tree-structured computation; the second one resembles a left linear computation tree. We shall argue that both translation schemes can guarantee an efficient execution depending on the function which is performed during iteration.

During horizontal transformation $sum\_by_{B,A}$ and $merge_A$ are combined by operator L. To translate their combination into an iterative program during vertical transformation, we generate the new function $sum\_merge_{B,A}$ which performs merging and aggregation simultaneously. The recursive form of the new function immediately leads to an iterative program form.

4.1. Horizontal Transformation

The horizontal transformation of

$(sum\_by_{B,A}\ (merge\_sort_A\ list))$

is based on the following theorem which provides a sufficient condition for distributing a function f_1 over the tree operator.

Theorem 1:

Let f_1 and f_2 be two functions and let l_1, l_2, l_3 be lists. If

$$(f_1\ (f_2\ l_1\ l_2)) = (f_1\ (f_2\ (f_1\ l_1)\ (f_1\ l_2)))$$

then

$$(f_1\ (tree\ f_2\ l_3)) = (tree\ (L\ f_1\ f_2)\ (map\ f_1\ l_3))$$

The inductive proof of the theorem can be found in [FREY85a]. Intuitively, the condition of the theorem requires f_1 and f_2 to be "almost" independent. Since the two functions do not commute with each other, f_2 follows each application of f_1 in a tree-like computation.

To apply Theorem 1 to the combination of functions $(sum\_by_{B,A}\ (merge\_sort_A\ list))$, which extends to

$(sum\_by_{B,A}\ (tree\ merge_A\ (map\ sort_A\ (convert\ list))))$

by the definition of $merge\_sort_A$, we need to show that functions $sum\_by_{B,A}$ and $merge_A$ satisfy the condition of the above theorem. Notice, that the input to both, $merge_A$and $sum\_by_{B,A}$, are lists *sorted* on attribute A. Informally, if $merge_A$ is applied first, the result is a sorted list with the same tuples as in the two input lists. Applying $sum\_by_{B,A}$ yields a sorted list containing one tuple for each value of A recording the sum of A's B values.

On the other hand, if we first apply $sum\_by_{B,A}$ to both sorted lists l_1, l_2, we obtain two sorted lists l'_1, l'_2, each containing one tuple for each A value recording the sum of all B values. Applying function $merge_A$ to l'_1 and l'_2results in a sorted list containing at most two successive tuples for each A value. Applying the aggregate function again yields the same aggregated list as for the first case.

As $sum\_by_{B,A}$ and $merge_A$ satisfy the condition, we can apply the theorem to the above expression, thus yielding the expression

$$(tree\ (L\ sum\_by_{B,A}\ merge_A)\ (map\ sum\_by_{B,A}\ (map\ sort_A\ (convert\ list))))$$

The operations specified by the nested subexpression $(map\ sum\_by_{B,A}...)$ are to sort all runs *before* applying the aggregate function $sum\_by_{B,A}$. However, we can avoid scanning the list of runs twice by performing the aggregate operation *immediately after* sorting the run. We therefore *fold* the nested map expressions into one map expression combining the sort and the aggregate function by operator L:

$$(tree\ (L\ sum\_by_{B,A}\ merge_A)\ (map\ (L\ sum\_by_{B,A}\ sort_A)\ (convert\ list)))$$

which represents the first result of horizontal transformation. The expression determines to first convert a list of tuples into runs before applying the sort and aggregation function to each individual run. Finally, all runs are merged and aggregated using a tree-like computation.

We continue the horizontal transformation on the combination of functions

$$(L\ sum\_by_{B,A}\ sort_A)$$

By the definitions of $sort_A$ and L in Section 3, we rewrite the combination into the expression

$$(sum\_by_{B,A}\ (tree\ merge_A\ list))$$

which allows us to apply Theorem 1 again. The resulting expression is

$$(tree\ (L\ sum\_by_{B,A}\ merge_A)\ \ (map\ sum\_by_{B,A}\ list))$$

As the application of $sum\_by_{B,A}$ to each tuple of the run is superfluous, the previous expression reduces to

$$(tree\ (L\ sum\_by_{B,A}\ merge_A)\ list)$$

which constitutes the second result of the horizontal transformation. As each merge is immediately followed by the aggregate function the final list of tuples consists of one tuple for each A value recording the sum of its B values.

We may now combine both expressions by substituting the latter for $(L\ sum\_by_{B,A}\ sort_A)$ into the former yielding the final expression of horizontal transformation:

$$(tree\ (L\ sum\_by_{B,A}\ merge_A)\ (map\ (tree\ (sum\_by_{B,A}\ merge_A))\ \ (convert\ list)))$$

4.2. Vertical Transformation

Since we are concerned with an fast execution of the programs which we generated in the previous subsection, we suggest to transform these high-level program specifications into iterative programs. The final expressions resulting from horizontal transformation show that we have to develop transformations for the *tree* operator, the *map* operator and the combination of functions $merge_A$ and $sum\_by_{B,A}$. Operator *map* simply translates into a loop. We develop a transformation for the more complex *tree* operator and the function combination in the next two subsections. Surprisingly, we can describe both transformations independently. Since the performed transformations are quite complex, we represent only the final iterative programs without going into the details of the transformation itself. The interested reader is referred to [FREY85a].

4.2.1. Transforming the Tree Operator into Iteration

At the beginning of this section we mentioned that the control structure operator *tree* has a natural recursive implementation. For fast execution we would like to replace the "tree recursion" by iteration. We derive two different iterative programs with different computational behavior [FREY85a]. The first iterative program maintains the tree structure by performing the desired computation in *rounds*, thus simulating the tree specification in a bottom up manner. Figure 3 shows the computational behavior of the first iterative program in case of f being the *merge* function.

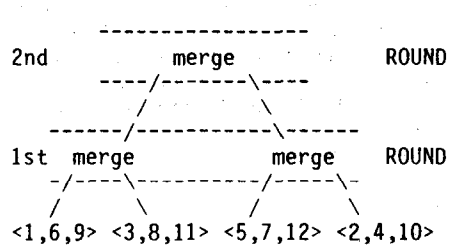

Figure 3: The Bottom-Up Computation in Rounds

During each round function f is applied to runs of the same length before the next round is encountered. If f is the *merge* function, then the iterative program exactly performs the computation as shown in Figure 2.

In contrast to the first iterative program the second one we generate performs a computation which resembles a left linear computation tree. We show the computation tree in Figure 4 for f being the *merge* function.

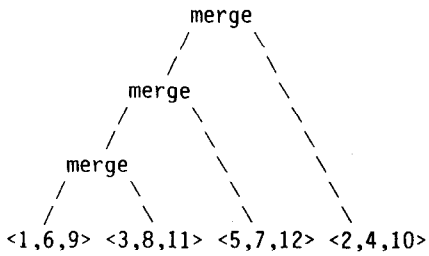

Figure 4: The Left Linear Computation

During each iteration function f is applied to one initial run and the "intermediate" result which has been accumulated so far. If f is the *merge* function, then the iterative program successively merges each run with a tuple list whose length increases with every merge. Clearly, the transformation has changed the $n \log n$ complexity of the initial tree specification into a program which has a n^2 complexity with n being the initial number of tuples.

Despite the disadvantage in this example there may exist cases where one would favor the second transformation. To justify our claim we examine in more detail the representation of tuple lists by relations in database systems. Usually, relations are stored on secondary storage. The DBMS needs to access relations in *memory* to perform operations necessary for the query evaluation. For this purpose, the systems maintains a *buffer* which keeps part of the relations in memory. Its contents is changed frequently by reading from and writing to secondary storage whenever the DBMS has finished its operations on one part and needs buffer space for processing other parts. As read and write operations are time consuming, they often dominate the overall query evaluation time. Decreasing the number of read and write operations usually improves the overall processing time.

Suppose that the buffer is large enough to store three initial runs. Let f be the combination of functions $sum\_by_{B,A}$ and $merge_A$. Furthermore, assume that the new run resulting from merging and applying aggregation has (approximately) the same length as the initial two input runs. The output run then still fits into the available buffer space. If the condition holds for every application of f, the second transformation turns out to be superior as far as read and write operations are concerned. During the execution of the second iterative program the intermediate run always resides in memory and is immediately available as input for the next application of $sum\_by_{B,A}$ and $merge_A$. In contrast, the iterative program generated by the first transformation writes out each run created in round i before reading it again in round $i+1$. More formally, if there are n initial runs which need to be processed by $sum\_by_{B,A}$ and $merge_A$, the first iterative program initiates $2n-1$ read and $n-1$ write operations[3] under the above assumptions. The program generated by the second transformation uses only n read operations thus evaluating the aggregate much faster.

The assumptions made above are realistic for the computation of aggregates. Often, the number of values, which determine the partition of tuples into subsets, is much smaller than the number of tuples in the relation. Consider the example introduced in the introduction of this paper. Assume that the relation EMP contains 10,000 tuples describing employees in 50 different departments. If each initial run contains more than 50 tuples *and* if it fits into one buffer block, the above conditions are satisfied. The initial sort, followed by the aggregate function, results in at most 50 tuples recording the partial sum for each department existing in that run. If the subsequent phase performs merging and aggregation using the second iterative program the execution resembles a computation with one "bucket" for each department always residing in memory.

4.2.2. Transforming Functions Sum_by and Merge into Iteration

The final expressions resulting from horizontal transformation contain the combination of functions

$$(L\ sum\_by_{B,A}\ merge_A)$$

Using the definition of L of Section 2, we can rewrite the expression into the more familiar form

$$(sum\_by_{B,A}\ (merge_A\ l_1\ l_2))$$

Based on the recursive definitions of both functions, the iterative program would consist of two successive loops, one performing the sort, the other performing aggregation. To avoid the intermediate sorted list, we would like to generate a program which consists of *one loop* to perform sorting and aggregation at the same time. Based on recursive definition functions $sum\_by_{B,A}$ and $merge_A$, we show in [FREY85a] how to generate a new function $sum\_merge_{B,A}$ from the initial function definitions. A nontrivial, laborious transformation yields the new function $sum\_merge_{B,A}$ which performs aggregation and sorting simultaneously. $(sum\_merge_{B,A}\ l_1\ l_2)$ which has the following recursive form:

$$\begin{array}{l}(if\ (empty?\ l_1)\ l_2\\ \quad (if\ (empty?\ l_2)\ l_1\\ \quad\quad (if\ (gr_A?\ (first\ l_1)\ (first\ l_2))\\ \quad\quad\quad (out\ (first\ l_2)\ (sum\_merge_A\ l_1\ (rest\ l_2)))\\ \quad\quad\quad (if\ (eq_A?\ (first\ l_1)\ (first\ l_2))\\ \quad\quad\quad\quad (out\ (add_B\ (first\ l_1)\ (first\ l_2))\ (sum\_merge\ l_1\ (rest\ l_2)))\\ \quad\quad\quad\quad (out\ (first\ l_1)\ (sum\_merge_A\ (rest\ l_1)\ l_2))))))\end{array}$$

The $sum\_merge_{B,A}$ program differs from the $merge_B$ program only in case the A values of the first elements in both lists are the same. The newly generated program performs the aggregation by function add_B instead of returning one tuple. Using the transformation schemes in [BURS77] or [COHE80], the recursive form of the new program immediately leads to the following iterative program:

```
result := *empty*;
WHILE not (empty? l1) and not (empty? l2) DO
  IF (gr_A? (first l1) (first l2)) THEN
    result := (out (first l2) result); l2 := (rest l2);
  ELSE
    IF (eq_A? (first l1) (first l2)) THEN
       result := (out (add_A (first l1) (first l2)) result);
       l1 := (rest l1); l2 := (rest l2);
    ELSE
       result := (out (first l1) result); l1 := (rest l1);
END_DO;

IF (empty? l1) THEN
  result := (out l2 result)
ELSE
  result := (out l1 result);
```

Comparing both forms for function $merge\_by_{B,A}$, the reader may

---

3 One may reduce the number of read and write operations slightly for the computation in rounds. However, the computation in rounds will never perform as well as the second iterative program.

convince him(her)self that the recursive form is much more suited for formal manipulation than the iterative version.

5. Generalization of Transformation

In the previous section we demonstrated horizontal and vertical transformation using the aggregate function $sum\_\_by_{B,A}$. In the following subsection we extend the transformation to other aggregate functions by introducing two new control structure operators. These are used to redefine all aggregate functions. The second subsection then summarizes all transformations by defining four transformation rules. We apply them to a final example.

5.1. Redefinition of Aggregate Functions

The major question is what needs to be changed for the transformation of other aggregate functions. Clearly, the horizontal transformation rests on the condition of Theorem 1. Do we have to repeat vertical transformation for every aggregate function separately? The analysis of other aggregate functions, such as "the minimum (maximum) B value for each value of attribute A" or "number of tuples having the same A value", reveals that only the *functional part* of the aggregation changes, that is the function applied while scanning the list of tuples. The control structure, that dictates *how to scan the list of elements*, always remains the same. This observation leads to the definition of a new *control structure operator*, $agg\_\_by_A$, which clearly separates the two aspects. Informally, $(agg\_\_by_A\ f_1\ l_1)$ with f_1 being a function of arity two and l_1 being a list implements a computation which "accumulates the final result by applying function f_1 to subsequent elements in the list l_1 as long as they have the same A values". More formally, we define function $(agg\_\_by_A\ f_1\ l_1)$ to be

```
(if (empty? l1) *empty*
  (agg_A f1 (first l1) (rest l1)))
```

where $(agg_A\ f_1\ ele\ l_1)$ is defined by

```
(if (empty? l1) (out ele l1)
  (if (eq_A? ele (first l1))
    (agg_A f1 (f1 ele (first l1)) (rest l1))
    (out ele (agg_A f1 (first l1) (rest l1)))))
```

The only difference between functions $agg\_\_by_{B,A}$ and $sum\_\_by_{B,A}$ is that the former has an additional parameter specifying the function f_1 which is applied during the scan of l_1.

Similarly, we generalize $(sum\_\_merge_{B,A}\ l_1\ l_2)$ by introducing function

$(agg\_\_merge_A\ f_1\ l_1\ l_2)$

which applies function f_1 to any two elements with the same A values while merging. This generalization again distinguishes between the control structure, used to scan the list of tuples, and the function performed during the scan. We define $(agg\_\_merge_A\ f_1\ l_1\ l_2)$ by

```
(if (empty? l1) l2
  (if (empty? l2) l1
    (if (gr_A? (first l1) (first l2))
      (out (first l2) (agg__merge_A f1 l1 (rest l2)))
      (if (eq_A? (first l1) (first l2))
        (out (f1 (first l1) (first l2))
          (agg__merge_A f1 l1 (rest l2)))
        (out (first l1) (agg__merge_A f1 (rest l1) l2))))))
```

In [FREY85a] we show that the combination of functions $(agg\_\_by_A\ f_1\ (merge_A\ l_1\ l_2))$ translates into function $(agg\_\_merge_A\ f_1\ l_1\ l_2)$ by exactly the same transformation as for the specialized case $(sum\_\_by_{B,A}\ (merge_A\ l_1\ l_2))$.

Using these more general control structure operators, we redefine $(sum\_\_by_{B,A}\ list)$ by

$(agg\_\_by_A\ add_B\ list)$

where add_B is the *addition* function specialized to add the B values of two elements in the list. The new definition of $sum\_\_by_{B,A}$ nicely shows the distinct use of attributes A and B. Values of attribute A distinguish the different subsets on which to perform the aggregation. Attribute B determines which values to aggregate on.

We may define other aggregate functions in a similar way. For instance, consider the functions $(max\_\_by_{B,A}\ list)$ and $(min\_\_by_{B,A}\ list)$, which compute the maximum and minimum B value for each A value in the *list*, respectively. We redefine both functions by $(agg\_\_by_A\ max_B\ list)$ and $(agg\_\_by_A\ min_B\ list)$ with functions max_B and min_B returning the tuple with the larger and smaller B value, respectively.

The aggregate function $(count_A\ list)$, which counts the number of elements for each A value in the list is implemented by

$(agg\_\_by\ add_{CNT}\ list)$

where CNT is a new attribute whose initial value is 1. Similarly, duplicate elimination is defined by

$(agg\_\_by_A\ one\ list)$

with A being the list of all key attributes and with function $(one\ t_1\ t_2)$ returning either tuple.

The aggregate function $(average\_\_by_{B,A}\ list)$, which computes the average B value for each A value, needs some more attention. As the average computation is based on counting and summation, we need to compute the number of tuples for each A value and the sum of B values for each A value by $(sum\_\_by_{B,A}\ list)$ and $(count\_\_by_{CNT,A}\ list)$ before dividing the sum by the count value. If we rewrite both functions by their aggregate definitions $(agg\_\_by_A\ add_B\ list)$ and $(agg\_\_by_A\ add_{CNT}\ list)$ we recognize that both functions *access the same list using the same control structure.* Instead of accessing the *list* twice, we may apply both aggregate functions in "parallel" during one scan, thus leading to the expression

$(agg\_\_by_A\ \{add_B\ add_{CNT}\}\ list)$

The { } notation is used to denote the simultaneous application of a list of functions while scanning the *list*. Goldberg et al. call the proposed transformation *horizontal loop fusion* [GOLD84].

5.2. Transformation by Rules

To describe the translation of aggregate functions more formally, we introduce *four transformation rules* which summarize the transformation results of the previous subsections. A transformation rule has the form $(t_1 \rightarrow t_2)$ which specifies to replace an occurrence of expression t_1 by expression t_2. For our transformation we define the following rules with f_1, f_2 being functions:

Rule I: Combination Rule
$(\{(agg\_\_by_A\ f_1\ list)\ (agg\_\_by_A\ f_2\ list)\} \rightarrow$
$(agg\_\_by_A\ \{f_1\ f_2\}\ list))$

Rule II: Tree Rule
$((f_1\ (tree\ f_2\ list)) \rightarrow (tree\ (L f_1 f_2)\ (map\ f_1\ list)))$

Rule III: Fusion Rule
$((map\ f_1\ (map\ f_2\ list)) \rightarrow (map\ (L f_1 f_2)\ list))$

Rule IV: Aggregate-Merge Rule
$((L\ (agg\_\_by_A\ f_1)\ merge_A) \rightarrow (agg\_\_merge_A\ f_1))$

The first rule allows to combine several aggregate functions which scan the same list using the same control structure. Rule II reflects the result of Theorem 1. We can only apply the rule if the condition of the theorem is satisfied. Rule III combines two loops into one by applying both functions simultaneously, thus avoiding the creation of an intermediate list. The last rule replaces the functional combination of $agg\_\_by_A$ and $merge_A$ by the newly derived function $agg\_\_merge_A$.

We demonstrate the transformation by rules using the SQL query

SELECT AVG(Salary) FROM EMP
GROUP BY Department

which computes the average salary for each department in the EMP relation.

Before we can apply the rules in the given order, we translate the query into the internal form

$(average\_\_by_{Sal.,Dep.}\ (merge\_\_sort_{Dep.}\ EMP))$

In the next step we decompose the the average aggregation into the aggregate functions $sum\_\_by_{Sal.,Dep.}$ and $count_{Dep.}$ followed by function $DIV_{Sal.,CNT}$ which divides the sum and the count for each department value. Using the above definitions for both aggregate functions and the *sort__merge* function we obtain the expression

$(DIV_{Sal.,CNT}$
$\{(agg\_\_by_{Dep.}\ add_{Sal.}$
$(tree\ merge_{Dep.}\ (map\ sort_{Dep.}\ (convert\ EMP))))$
$(agg\_\_by_{Dep.}\ add_{CNT.}$
$(tree\ merge_{Dep.}\ (map\ sort_{Dep.}\ (convert\ EMP)))) \})$

Applying Rule I once yields the expression

$(DIV_{Sal.,CNT}$
$(agg\_\_by_{Dep.}\{add_{Sal.}\ add_{CNT}\}$
$(tree\ merge_{Dep.}\ (map\ sort_{Dep.}\ (convert\ EMP)))))$

Using Rule II and Rule III for the next transformation steps, we generate

$(DIV_{Sal.,CNT}$
$(tree\ (L\ AGG\_\_BY_{Sal.,CNT,Dep.}\ merge_{Dep.})$
$(map\ (L\ AGG\_\_BY_{Sal.,CNT,Dep.}\ sort_{Dep.})\ (convert\ EMP))))$

with $AGG\_\_BY_{Sal.,CNT,Dep.}$ being equal to

$(agg\_\_by_{Dep.}\{add_{Sal.}\ add_{CNT}\})$

We apply Rule IV to derive

$(DIV_{Sal.,CNT}$
$(tree\ (agg\_\_merge_{Dep.}\{add_{Sal.}\ add_{CNT}\})$
$(map\ (L\ AGG\_\_BY_{Sal.,CNT,Dep.}\ sort_{Dep.})\ (convert\ EMP))))$

which represents the final result of the rule-based transformation. The expression defines the following order of operations: first, convert the list of tuples into runs, then apply sorting and aggregation to each of them. A tree-like computation produces one list of tuples each of which contains the number of tuples and the salary sum for each department. Using the definitions of $AGG\_\_BY_{Sal.,CNT,Dep.}$ and the $sort_{Dep.}$, we may complete the translation by the same transformation as shown in Subsection 4.1.

To guarantee fast execution we could continue the transformation of the tree expression into an iterative program based on the definitions for operators *map* and *tree* as described in Subsection 4.2.

6. Conclusion

We presented a improvement for the computation of aggregate functions which we derive by successive transformation by rules. We showed that independently defined programs for the aggregate and the sort functions can be translated systematically into a more efficient program which performs aggregation while sorting [KLUG82]. We generalize the translation to other aggregate functions as well by introducing a uniform notation which allows us to specify rules for their transformation.

In subsection 4.2.1. we compared the computational behavior of the two programs generated. To demonstrate the advantage of transformation, one would like to compare all three programs, i.e. the initial program which performs aggregation after sorting and the two derived programs, on experimental results. We did not perform such experiments to compare their performances. However, the results of Bitton et al. who use the idea of eliminating duplicates while sorting immediately applies to the computation of aggregates in rounds [BITT83].

7. Acknowledgement

We wish to thank Laura Haas for carefully reading a draft of this paper. While the first author was a student at Harvard University, this work was supported by the Office of Naval Research under grant ONR-N00014-83-K-0770.

Bibliography

[ASTR76] Astrahan, M. et al., *SYSTEM R: Relational Approach to Database Management,* ACM Transactions of Database Systems **1,2** (June 1976) pp. 97-137.

[BACK78] Backus,J., *Can Programming be liberated from the von Neuman Style? A Functional Style and its Algebra of Programs,* Communications of the ACM **21,8** (August 1978) pp. 613-641.

[BELL84] Bellegarde, F., *Rewriting Systems on FP Expressions that Reduce the Number of Sequences they Yield,* ACM Symposium on LISP and Functional Programming (August 1984) pp. 63-73.

[BIRD84] Bird, R.S., *The Promotion and Accumulation Strategies in Transformational Programming,* ACM Transactions on Programming Languages and Systems **6,4** (October 1984) pp. 487-504.

[BITT83] Bitton, D. and DeWitt, D.J., *Duplicate Record Elimination in Large Data Files,* ACM Transactions of Database Systems **8,2** (June 1983) pp. 255-265.

[BURS77] Burstall, R,M. and Darlington, J., *A Transformation System for Developing Recursive Programs,* Journal of the ACM **24,1** (January 1977) pp. 44-67.

[COHE80] Cohen, N.H., *Source-to Source Improvement of Recursive Programs,* PhD Thesis, Harvard University (May 1980).

[DARL76] Darlington, J. and Burstall, R.M., *A System which Automatically Improves Programs,* Acta Informatica **6,1** (January 1976) pp. 41-60.

[FREY85a] Freytag, J.C., *Translating Relational Queries into Iterative Programs,* PhD Thesis, Harvard University, also Technical Report TR-14-85 (September 1985).

[FREY85b] Freytag, J.C., *Rule-Based Translation of Relational Queries into Iterative Programs,* IBM Research Report RJ 4974; also Proceedings SIGMOD '86 Washington, D.C. (May 1986) (December 1985).

[GOLD84] Goldberg, A. and Paige, R., *Stream Processing,* ACM Symposium on LISP and Functional Programming (August 1984) pp. 53-62.

[HUET80] Huet, G., *Confluent Reductions: Abstract Properties and Applications of Term rewriting Systems,* Journal of the ACM **27,4** (October 1980) pp. 797-821.

[JARK84] Jarke, M. and Koch,J., *Query Optimization in Database Systems,* ACM Computing Surveys **16,2** (June 1984).

[KLUG82] Klug, A., *Access Paths in the Abe Statistical Query Facility,* ACM SIGMOD Conference on Management of Data (1982).

[KNUT73] Knuth, D.E., *The Art of Computer Programming Vol. 3,* Addison Wesley, Reading, MA (1973).

[STON76] Stonebraker, M. and Wong, E., et al., *The Design and Implementation of INGRES,* ACM Transactions of Database Systems **1,3** (September 1976) pp. 189-222.

[WILL82] Williams, J.H., *Notes on the FP Style of Functional Programming,* J.Darlington, P. Henderson, D.A. Turner, *Functional Programming and its Application,* Cambridge University Press (1982) pp. 193-215.