Inform. Systems Vol. 10, No. 3, pp. 325–330, 1985
Printed in the U.S.A.

0306–4379/85 $3.00 + .00

DECOMPOSITION OF SELECT EXPRESSIONS

G. A. BLAAUW, A. J. W. DUIJVESTIJN and F. NIEUWERTH
Twente University of Technology, Dept. of Computer Science, P.O. Box 217, 7500 AE Enschede, Netherlands

(*Received* 19 *July* 1983; *in revised form* 5 *June* 1984)

Abstract—A select operation that is part of an expression applying to a relational database is decomposed into one or more independent select operations for the purpose of optimising the relational expression. The select expression is treated as a logical expression. From the canonical form of this expression an optimal conjunctive form is obtained which can be decomposed into separate select operations. These separate selects can then be moved to the most effective place within the relational expression. The method also eliminates redundancy in the original expression. A prototype has been used in developing the optimisation method; from this prototype an implementation for use in an actual system has been derived.

Keywords: information systems, relational databases, optimisation of relational expressions, select expressions.

INTRODUCTION

For the relational database model, originally proposed by Codd[1], several optimisation methods have been proposed. This paper concerns the optimisation on the highest implementation level, where an expression that is used to access the data is rewritten in a form that is more efficient for a given model of access path selection. In a separate paper a general method for such an optimisation is presented[2]. The decomposition of select expressions, which is the subject of the present paper is intended to precede that kind of optimisation and to make it more effective.

The manipulation of logical expressions needed in this application is similar to the simplification of algebraic expressions familiar in formula manipulation. Also in compiler optimisation, related problems arise. Hall and Todd[2] treat this problem by using the theory of semirings. In our paper we use methods originating from digital switching theory; these prove to lead to a relatively simple algorithm.

Because the names of relational operations, such as "select," are used with different semantics, we start by giving some definitions. This also allows us to state the aim and the essence of our method more precisely. We next present the main steps of the method: parsing, transformation to logical expression, and decomposition. The method has been embodied in an executable prototype. This prototype has demonstrated the effect of the decomposition and has been used as a definition from which an implementation for an actually produced database system has been made.

DEFINITION

The *select* operator SL is a monadic operator on a relation (or table) r. The select contains a *select expression* $F(X)$ which uses a subset X of the attributes (or columns) $R(r)$ of r as operands and results in a boolean value. $SL(r,F)$ is the set of tuples (or rows) u of r for which F has a true result; the other tuples of r do not appear in the result of the select. When the table r is implied we denote the select as $SL\ F$.

AIM

When the rows of the table r are made available to the select operator the execution of the select operation requires only compute time to evaluate the expression F. This time is normally far less than the time involved in obtaining the rows of the table from secondary storage; it also is far less than the time required by most of the other relational operations, which again involve secondary storage access. Therefore, the select is considered an inexpensive operation in comparison to other relational operations. Furthermore the select never increases the size of a table; it often decreases that size substantially. The optimisation methods that reorder the operations of a relational expression consequently try to place the select as early as possible, that is, as early as the attributes X used in the expression $F(X)$ allow.

The aim of the decomposition method is to split a select expression in parts that jointly are equivalent to the given select expression, but which individually may be placed more advantageous in the relational expression than is possible for the given select expression as a whole. As an example the select expression A OR (B AND C) requires the three boolean attributes, A, B, and C, to be available. According to the distributive law this expression can be rewritten as (A OR B) AND (A OR C). Hence the original select is equivalent to two selects, with expressions A OR B and A OR C, applied in succession. Each of these selects requires only

two attributes and could possibly be placed earlier than the original select. If so, the tables of the relational expression can be reduced earlier, which gives a more optimal evaluation.

This particular method of optimising select operations thus consists of two parts. First, we decompose a select as a sequence of selects; second, we give an optimum place to these selects. The second action occurs as part of an overall optimisation method and is described elsewhere[3]. The first action is the subject of this paper.

METHOD

To break the select apart in several selects, the select is parsed into subexpressions that yield a boolean result and are connected by logical operators. These subexpressions are then treated as logical variables and the select expression is considered a logical function of these variables. The select expression is then transformed to the minimal conjunctive form for this logical function, which consist of a number of terms separated by AND operators. Each of these terms is then used as the select expression for a separate select operation. Some of the selects in which the original is decomposed may be redundant. But "'redundant" does not necessarily mean useless. Therefore, as a last step we decide whether to retain the redundant selects or not.

PARSE

Subexpressions

The first purpose of the parser of the select expression is to recognise the smallest subexpressions of the select expression that still yields a boolean result. Such subexpressions may be boolean variables and constants as well as comparisons, equalities, and inequalities yielding a boolean result. The complexity of this first task depends upon the nature of the syntax of the language in which the select expression is written. If necessary the syntax may be simplified such that the parse becomes simpler at the expense of treating some subexpressions as an entity, even though they could be subdivided.

Identification

The second purpose of the parser of the select expression is to identify the subexpressions and associate these with logical variables. In particular, subexpressions whose net effect is identical or the exact inverse should be treated as the same logical variable. Again, there is freedom to stop short of the theoretical ideal. For instance, it is very questionable whether the parser should spend time and space to identify $A = 2 \times B$ as identical to $B = A - B$. In the prototype that kind of identity was not recognised but the prototype did recognise $A<B$ as the inverse of $B \leq A$.

The details are now considered in turn.

SYNTAX

The parser analyses boolean expressions generated by the following grammar, where the nonterminal "bool expr" is the initial symbol. The notions ending with "-symbol" are terminal symbols. Some of the terminal symbols are given by their representation. The metanotions CHAIN, SEQUENCE, OPTION and PACK are taken from the ALGOL 68 definition.

```
bool expr: secondary-CHAIN-bool operator.
secondary: primary; not-symbol, primary.
primary: bool value; subexpression; bool expr-PACK.
subexpression: bool column name;
                    value, compare operator, value.
value: column name; arithmetic expression; constant.
arithmetic expression: {, text, }.
text: character-SEQUENCE.
bool operator: and-symbol; or-symbol.
compare operator: <; ≤; >; ≥; =; ≠.
column name: character c-symbol, digit-SEQUENCE.
bool column name: character b-symbol, digit-SEQUENCE.
digit: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9.
character: "any character except { and }".
constant: sign-symbol-OPTION,
          digit-SEQUENCE,(point-symbol,digit-SEQUENCE)-OPTION.
          character b-symbol: B.
          character c-symbol: C.
          bool value: true-symbol; false-symbol.
          true-symbol: 1.
          false-symbol: 0.
```

The language generated by this grammar is denoted by BL1.

LANGUAGE TRANSFORMATION

A sentence belonging to BL1 is transformed into another sentence by changing the productions originating from the production rule:
subexpression: bool column name; value, compare operator, value. in the following way.

value 1 < value 2 => value 1 < value 2 or value 2 > value 1,
value 1 ≥ value 2 => NOT(value 1 < value 2) or
NOT(value 2 > value 1),
value 1 > value 2 => value 1 > value 2 or value 2 < value 1,
value 1 ≤ value 2 => NOT(value 1 > value 2) or
NOT(value 2 < value 1),
value 1 = value 2 => value 1 = value 2 or value 2 = value 1,
value 1 ≠ value 2 => NOT(value 1 = value 2) or
NOT(value 2 = value 1),

The choice between value 1 < value 2 and value 2 > value 1 is made by considering the productions "value 1 < value 2" and "value 2 > value 1" as character strings. The well-known ordering relation of character strings is used to determine this choice. If the character string "value 1 < value 2" is smaller than "value 2 > value 1" then value 1 < value 2 is chosen otherwise value 2 > value 1 is chosen. The other choices are made in a similar way. The transformed sentence is an element of a language BL2.

TRANSFORMATION TO LOGICAL EXPRESSION

Subexpressions in BL2 are considered different if their corresponding character strings are different. To each subexpression a logical variable is associated. In the sentence of BL2 the subexpressions are replaced by their logical variables. In this way a logical expression is obtained. The logical expression is a logical function whose variables can take either the value true or false. True is represented by 1 and false by 0. The logical expression consists of either one term or a number of terms separated by an OR operator. A term consists of either one factor or a number of factors separated by either an AND operator or an AND followed by a NOT operator. A factor is a logical variable or a boolean value.

LOGICAL TRANSFORMATION

The logical function specified by the logical expression can be given by a truth table in which the function is tabulated.

Truth table

A logical function can be specified by tabulating the function as a so-called *truth table*. A truth table is the tabulation of all possible combinations of values taken by the variables of the logical expression. Thus the example given earlier contains three variables, A, B, and C, that each can take either the value true or false. By tabulating the output of the expression A OR (B AND C) for all possible input values, the truth table of Fig. 1 is obtained.

The truth table can be condensed by giving only the rows for which the function value is true. By giving the decimal equivalent of each of these rows treated as a binary number, we can write the truth table even more compactly. For the foregoing example the truth table then becomes 3 4 5 6 7.

When the specification is given in a Karnaugh diagram[4], as in Fig. 2, we can visually observe the binary encoding. The Karnaugh diagram is akin to the Venn diagram. It displays the relation between the variables and the codes which we use in the decomposition.

Observe that Figs. 1 and 2 no longer contain the particular structure of the original logical expression, or its equivalent after applying the distributive law. Therefore, the truth table and the associated information about its variables is a general and neutral starting point for the decomposition process.

The logical function can also be represented by its so-called *canonical form* which consists of the terms that correspond to all rows of the truth table for which the logical function is true. For the given example the canonical form is (A' AND B AND C) OR (A AND B' AND C') OR (A AND B' AND C) OR (A AND B AND C') OR (A AND B AND C) where A' stands for NOT A.

	A	B	C	value
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

Fig. 1. Truth table for A OR (B AND C).

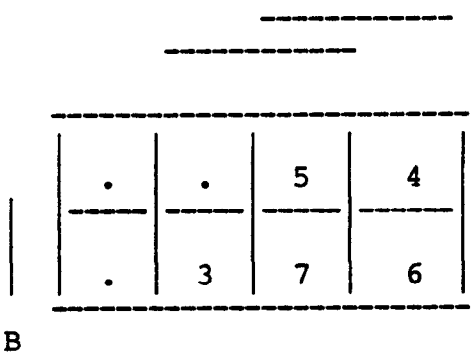

Fig. 2. Karnaugh diagram of Fig. 1.

Prime implicants

The number of terms and the total number of factors that appear in all the terms of the canonical form can be reduced by finding the prime implicants of the given expression, as is standard practice for logical minimalisation. This reduction uses the theorem (x AND y) OR (x AND y') = x which is based on the following postulates of boolean algebra:

distributive law:	x AND y OR x AND z = x AND (y OR z)
complement law:	x OR x' = 1, with 1 = true
identity law:	x AND 1 = x.

Hence two terms that differ in only one factor (two *adjacent* terms) can be combined to one term with one factor less.

A Boolean function f implies a function g if for every v satisfying $f(v)$ = true also $g(v)$ = true, where v is the complete set of variables occurring in f.

An *implicant* of a logical function f is a term that implies f.

A term $t1$ *subsumes* a term $t2$ if all the literals of $t2$ are contained in $t1$.

A *prime implicant* of a given logical function f is an implicant of f such that no other term subsumed by it is an implicant of f.

In the example of Fig. 2 the prime implicants are:

B AND C	(3 7)
A	(4 5 6 7)

Hence, we can replace the canonical solution of 5 terms and 15 factors with the prime implicant solution of 2 terms and 3 factors.

In the given example all prime implicants are necessary for the solution. This is not true in general. Therefore, after finding the complete set of prime implicants a minimal cover of these is selected.

The algorithm to determine the prime terms is based on Quine[6].

The prime terms may overlap. This means that they may have canonical terms in common. For example: (3 7) and (4 5 6 7) both satisfy 7.

Minimal conjunctive form

For the function specified by the truth table we have obtained a minimal disjunctive form of "AND" terms, or "min terms" separated by OR operators analogous to the methods used in digital switching theory[5]. The minimal conjunctive form of "OR" terms, or "max terms" separated by AND operators can be derived from the minimal disjunctive form.

We can obtain the conjunctive form from the disjunctive form by applying the principle of *duality*. This principle states that for a given expression an inverse expression can be derived by replacing true by false, false by true, AND by OR, and OR by AND. A simple application of duality is de Morgans Theorem stating that NOT(A AND B) is the same as (NOT A) OR (NOT B). By applying duality twice the original is obtained again.

In principle the disjunctive and conjunctive form are equally easily obtained. In practice, however, the disjunctive form is more readily visualised. Therefore, we use it as an intermediate step in obtaining the conjunctive form. We briefly outline the basic steps to be taken, using the example of Figs. 1 and 2, and refer to the literature[5] for a more thorough treatment.

Since we eventually want a conjunctive form we apply duality and start with the disjunctive form for the inverse of the desired logical function. For the given example the logical function is specified as 3 4 5 6 7 and the inverse is 0 1 2 as shown in Fig. 3.

The canonical disjunctive form is the OR of several terms that each contain as factors only the variables or their inverse separated by AND operators.

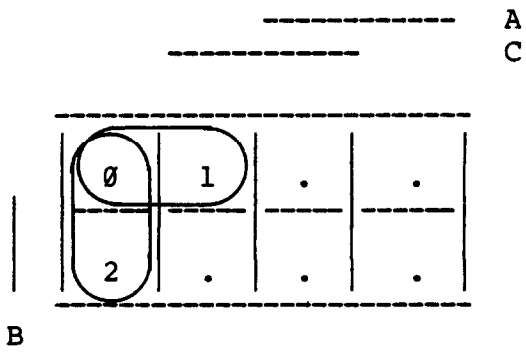

Fig. 3. Inverse of the function of Fig. 2.

Each variable occurs only once as a factor in a canonical term. The canonical form for our example can now be represented by the series of terms (0′ AND 1′ AND 2′) OR (0′ AND 1′ AND 2) OR (0′ AND 1 AND 2′), where 0′ means NOT 0, and the variables *A*, *B*, and *C* are represented by 0, 1, and 2.

The number of terms and the total number of factors that appear in all the terms of the canonical form can be reduced by finding the prime implicants of the given expression. The prime implicants are (0′ AND 1′) OR (0′ AND 2′). In the Karnaugh diagram of Fig. 3 the prime implicants are indicated by an oval. The adjacency of terms is displayed by adjacency of position (perhaps across the boundary) in the diagram.

From the minimal disjunctive form, such as (0′ AND 1′) OR (0′ AND 2′), the minimal conjunctive form is now obtained by applying duality; in this case we obtain (0 OR 1) AND (0 OR 2).

Each "OR" term can now be assigned to a separate select operation. Thus, for our example the two 'OR' terms of the minimal conjunctive form (*A* OR *B*) and (*A* OR *C*) are the select expressions of two select operations. These selects are then placed in an optimal position by the global optimisation algorithm.

Redundancy

In the given example all prime implicants are necessary for the solution. This is not true in general. Some of the terms may be redundant and could be omitted if desired.

As another example consider the select:

$$SL[(c3 > c2) \text{ AND } (c4 = c5)] \text{ OR } [(c6 \neq c7) \text{ AND } (c5 \neq c4)]$$

applied to the result of the join T4567 JN T2367, as illustrated in Fig. 4.

The attributes of the various tables are numbered. The subexpressions $c2, \ldots, c7$ use the corresponding attributes $2, \ldots, 7$, for instance $c2$ may be 5 × (attribute 2) + 7; table T4567 contains attributes 4, 5, 6, 7 and table T2367 contains attributes 2, 3, 6, 7. In its original form the select must follow the join since all six attributes participate in the select expression.

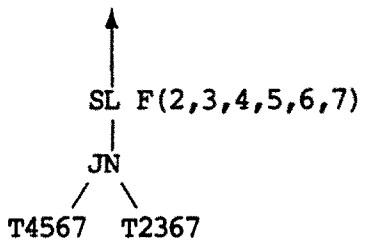

Fig. 4. Example of select in an expression prior to decomposition.

The parser recognises three logical subexpressions: $c2 < c3$, $c4 = c5$, and $c6 = c7$, which appear as the logical variables 0, 1, and 2 in the subsequent analysis. In terms of these variables the corresponding logical expression becomes:

(0 AND 1) OR (1′ AND 2′); the function value is 0 4 6 7 as shown in Fig. 5; the dual function value is 1 2 3 5; its min terms are 0′ AND 1, 0′ AND 2, and 1′ AND 2, as shown in Fig. 6; the dual max terms are: 0 OR 1′, 0 OR 2′, and 1 OR 2′; the corresponding selects *F*1, *F*2, and *F*3 have the expressions ($c2 < c3$) OR NOT ($c4 = c5$), ($c2 < c3$) OR NOT($c6 = c7$), and ($c4 = c5$) OR NOT($c6 = c7$).

The first select remains on the top of the join, the second select can be placed in the right branch of the join, the third select can be placed in the left branch of the join as shown in Fig. 7.

Logical analysis shows that because of the consensus theorem the second max term, 0 or 2′, is redundant. Hence, this max term and its corresponding select ($c2 < c3$) OR NOT($c6 = c7$) can be eliminated. The example demonstrates, however, that this redundant select *SL F*2(2367) can be placed low in the expression tree, which makes it potentially effective. Therefore, this select is retained. If, in contrast, the join were T2345 JN T4567, the second select would be placed on top of the join and since it is redundant, it might as well be eliminated.

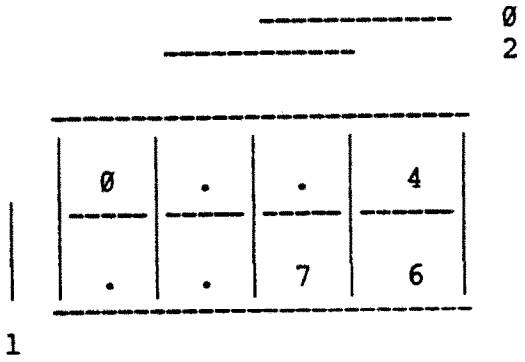

Fig. 5. Karnaugh diagram for select function of Fig. 4.

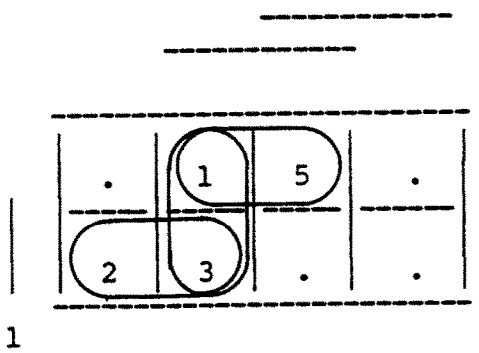

Fig. 6. Karnaugh diagram for the dual function of Fig. 5.

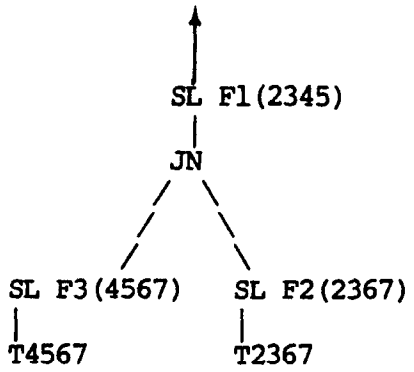

Fig. 7. Select of Fig. 4 after decomposition and relocation.

PROTOTYPE

the feasibility of the select optimisation has been demonstrated by a prototype written in APLDL[7]. The prototype gives a precise and complete description of the method concerned. As such it contains the essential algorithms. Because it is an executable description, the method can be demonstrated and tested for accuracy, consistency and effectiveness.

Thus the parser in this prototype recognises logical and compare operators, such that it can transform $c5 \neq c4$ to NOT($c4 = c5$), as illustrated in the second example. The relative desirability of recognising this kind of detail and its corresponding cost can be established via the prototype.

The prototype is an architectural description and is not concerned with implementation or data representation. These matters can, however, readily be deduced from the prototype by an experienced implementer. Thus, the prototype constitutes an important milestone in the management of a design. It assures the correctness of a major part of the design and allows review and feedback prior to the implementation effort.

The original prototype contained 250 lines of APL statements, including function headers and comments. A second prototype was developed to close the gap between the architectural specification and the ultimate implementation. This second prototype gave algorithms intended for the implementation language. This prototype took 500 lines of APL statements. From this second prototype the implementation was derived in a straightforward mapping. Three thousand implementation language statements were written which generated 10000 bytes of machine code[8].

EVALUATION

The decomposition method as described proves to be quite effective. It is able to decompose any select expression and eliminates redundancy in the process. The fact that decomposed terms can be placed low in the expression tree can be used to reduce the size of the tables early in the execution, it also may allow a more effective access method.

The use of a logical minimisation has the advantage of exploiting the efficiency of the logical operands and operations. Moreover, it can use well-established methods to their best advantage.

The decomposition method presented in this paper was implemented and incorporated in an existing system using the prototype. The implementation effort took about half the normal time because of the use of a prototype.

Acknowledgements—This study was performed under the direction of Ir. A. J. Du Croix, development manager at IBM International Operations Uithoorn, Netherlands. The implementation was performed under the direction of Ir. F. van der Landen of the IBM Uithoorn laboratories. The prototype was developed at the Technological University Twente, Enschede, Netherlands. The authors wish to thank Mr. J. Schoonenberg, director of IBM International Operations Uithoorn, management, and co-workers for wholehearted support.

REFERENCES

[1] E. F. Codd: A relational model of data for large shared data banks. *Communications of the ACM* **13**(6), 377–387 (June 1970).
[2] G. A. Blaauw, A. J. W. Duijvestijn and R. A. M. Hartmann: Optimisation of relational expressions using a logical analogon. *IBM Journal of Research and Development* **27** (5), 497–518. (Sept. 1983).
[3] P. A. V. Hall and S. J. P. Todd: Factorisations of algebraic expressions. Rep. UKSC 0055, IBM UKSC, Peterlee (1974).
[4] M. Karnaugh: The map method of synthesis of combinatorial logic circuits. *Transactions of the AIEE* **72**(part I), 593–598 (Nov. 1953).
[5] E. J. McCluskey: Minimization of boolean functions. *Bell System Technical Journal* **35**, 1417–1444 (November 1956).
[6] W. V. Quine: The problem of simplifying truth functions. *Amer. Math. Monthly* **59**, 521–531 (Oct. 1952).
[7] G. A. Blaauw, A. J. W. Duijvestijn and A. Ledeboer: An APL design language. Internal report IBM laboratories Uithoorn (May 1979).
[8] F. Nieuwerth: Optimisation of expressions in relational databases. Masters thesis, Electrical Engineering, Technological University Twente, Enschede, Netherlands, (Apr. 1983).