G. A. Blaauw
A. J. W. Duijvestijn
R. A. M. Hartmann

Optimization of Relational Expressions Using A Logical Analogon

An expression applying to a relational database is optimized by mapping the expression upon set expressions which, in turn, are transformed into logical expressions. These logical expressions then are optimized, taking into account the constraints that are inherent in relational expressions and the costs of those expressions. Subsequently a reverse transformation to relational expressions is applied. The method is developed for the traditional relational operators and is applicable to a variety of cost criteria. Common subexpressions as well as redundant expressions are optimized. A new relational operation "split" *is proposed that may be used effectively in an optimized expression. Results obtained with a model for the optimization method are presented.*

1. Introduction

Since the original proposal of the relational database model by Codd in 1970 [1], this method of organizing data has been studied extensively and is being applied in an increasing number of systems [2, 3]. The relational database is conceptually general and simple. Yet this generality gives it an initial performance disadvantage in comparison to earlier database designs, such as the hierarchical and the network approaches. A good part of this performance disadvantage, however, can be eliminated by the use of optimization in the implementation of the database. This optimization can take place at any of several implementation levels. In this paper we are concerned with the highest of these levels, where the 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. This rewritten expression uses the relational operators and tables that are available to the user of the database, as well as a few derived operators that are on the same level but commonly not available to the user. This method of rewriting has been studied extensively. The methods presented in the literature, however, are limited by the relational operators that can participate in the optimization [4] and by constraints with regard to adjacency of these operators [5–8]. Such restrictions make these methods only applicable locally within an expression. In contrast, the method presented in this paper is globally applicable. All normal relational operators are taken into account without any restriction concerning their adjacency.

The optimization method is based on the association of relational operations with set operations and subsequently on the association of set operations with logical operations. The latter association is well known, but the association of relational operations with set operations is documented only for a limited operator set [4, 9]; yet it is worth considering this association for the full operator set. Thus, we show that one operation can be expressed in terms of another, a property that is used in the optimization process.

The proposed method is in part heuristic and uses generally applicable logical transformations. Although we illustrate it with respect to a certain collection of constraints derived from a particular implementation, other conditions could be applied. Furthermore, many methods of logical manipulation are known, and they are readily adapted to the requirements of relational optimization.

To obtain a common basis of understanding and notation we start with a brief review of the theory of the relational

database model and describe the operators that are available for this model. Since optimization attempts to reduce the cost of the implementation, we next describe a specific method of implementing the database model with a row-by-row treatment for which the optimization is intended. For this implementation the cost of obtaining intermediate and final results of an expression is determined by the type of access to the tables and the size of the tables. The optimization itself depends upon the definition of universal relations, which are used in mapping the relational operations upon set operations and subsequently upon logical operations. The equivalent logical expression can then be placed in a canonical form and subsequently transformed into a closed cover of prime implicants. To combine common subexpressions a new relation "split" is introduced. Furthermore, the logical expressions may be transformed to satisfy the constraints of the corresponding relational operations and to minimize cost. The factors within the resulting terms are then ordered to further reduce cost. Finally, the logical expression is transformed back to a realizable relational expression. The method as presented has been embodied in an executable prototype which is used to demonstrate the effect of optimization for several examples.

2. Background and theory

• *Relations*

In the relational model the data item is the two-dimensional table called *relation.* The columns of the table are labeled by names that are called *attributes,* and the entries in each column are taken from a fixed set of values, called the *domain* of the attribute. The Cartesian product of domains $D_1, D_2, \cdots, D_k$ is the set of all k-tuples $(u_1, u_2, \cdots, u_k)$ such that u_1 is in D_1, u_2 is in D_2, $\cdots$, u_k is in D_k. A relation is a subset of the Cartesian product of the domains of its attributes. In a relation all tuples are different. Each tuple is called a *row* or an *entry*; the number of attributes is the *arity* of the relation; the number of tuples in a relation r is the *size* of r. If R is the set of attributes labeling the columns of a relation r, then r is said to be the *current relation* of R. R is called the *relation scheme* that defines the format of r, and the set of attributes is denoted by $R(r)$. Sometimes we use *table* for relation and *column* for attribute.

• *Functional dependency and key columns*

The values of entries in a relation often satisfy functional dependencies. By a *functional dependency* of Y upon X, written $X \rightarrow Y$, we mean that Y is determined by X, where X and Y are sets of attributes.

A tuple u taken from r with attributes X belonging to $R(r)$ is denoted by $u[X]$. Since all tuples are different, a relation satisfies functional dependency $X \rightarrow Y$ if and only if for all u_1 and u_2 in r, $u_1[X] = u_2[X]$ implies $u_1[Y] = u_2[Y]$.

If R is a relation scheme with attributes $A_1, A_2, \cdots, A_k$ and X is a subset of $A_1, A_2, \cdots, A_k$, we say that X is a *key* of R if

1. $X \rightarrow A_1 A_2 \cdots A_k$, and
2. For no proper subset Z of X is $Z \rightarrow A_1 A_2 \cdots A_k$.

There may be more than one key for a relation. Therefore, one of the keys may be designated as primary key. In this paper, however, we recognize just one key.

• *Relational operators*

Relational operators may either be monadic or dyadic. A monadic relational operator has one relation as argument, and a dyadic relational operator has two such arguments. In either case the result is again a relation. Relational operators define a relational algebra.

Project

The *project* operator PR is a monadic relational operator. It results in a new relation with attributes $Y = \{A_1, A_2, \cdots, A_m\}$ such that $Y \subseteq R(r)$. We define PR (r, Y) to be $\{u[Y] \mid u \in r\}$.

The project in general implies duplicate removal. The project can be replaced by a *remove column* operator RC, which eliminates columns but does not remove duplicates, and a subsequent *duplicate removal* operator DR, which removes duplicate rows.

Select

The *select* operator SL is a monadic operator on relation r and uses an expression F which uses the attributes of r as operands and results in a Boolean value. SL (r, F) is the set of tuples u of r for which F has a true result.

The attributes used by F are generally called *arithmetic attributes.* If we can write the expression as $B = F_1$, however, the attributes of F_1 are called the arithmetic attributes of F but the single attribute B of F is a *free attribute.*

Union

The *union* of two relations r and s is denoted by r UN s. The union is defined for relations with equal attributes. This means $R(r) = R(s)$ and hence $R(r) = R(r \text{ UN } s)$. The union of relations r and s is the set of tuples that are in r or s or both.

The result of the union should contain no duplicates. In the implementation, however, duplicates are allowed, provided they are subsequently removed. We exhibit these two steps by assuming that the output of UN has duplicates and that these are subsequently removed by a separate duplicate removal operator DR. We further distinguish the union operator UN from the *disjoint union* r DN s, for which it is known that the operands r and s have no tuples in common. Whereas the union UN in general creates duplicates, DN creates no duplicates and does not require DR.

Cartesian product

The *Cartesian product*, or *quad*, is a dyadic operator, written r QD s, where r and s are relations of arity k_1 and k_2, respectively, and their attributes are disjoint, i.e., $R(r) \cap R(s) = \varnothing$. The Cartesian product of r and s is the set of $(k_1 \times k_2)$ tuples u, such that $u[R(r)] = r$ and $u[R(s)] = s$.

Intersection

The *intersection* is a dyadic operator, written r IN s, where $R(s) \subseteq R(r)$ and where r IN s is the set of tuples u in r for which $u[R(s)]$ is in s.

Difference

The *difference* is a dyadic operator, written r DF s, where $R(s) \subseteq R(r)$ and where r DF s is the set of tuples u in r but with $u[R(s)]$ not in s.

Exclusion

The *exclusion* is a dyadic operator, writen r XC s, where $R(r) = R(s)$ and where r XC s is the set of tuples u in r but not in s, or u in s but not in r.

The exclusion operator can be expressed by other operators, e.g., r XC $s = (r$ UN $s)$ DF $(r$ IN $s)$.

Join

The *join*, also known as *natural join* [7], is a dyadic operator, written r JN s, where $R(r$ JN $s) = R(r) \cup R(s)$ and where r JN s is the set of tuples u with attributes $R(r) \cup R(s)$ such that there exist tuples u_1 in r and u_2 in s for which $u_1[R(r)] = u[R(r)]$ and $u_2[R(s)] = u[R(s)]$.

We assume $R(r) \cap R(s) \neq \varnothing$, to distinguish the join from the quad. Similarly, we assume $R(s) \not\subseteq R(r)$, to distinguish the join from the intersection.

Calculate

The *calculate* is a monadic operator, written CL $(r, X \leftarrow F)$. Calculate uses an assignment consisting of a left-hand side specifying an attribute X and a right-hand side consisting of an arithmetic expression F involving attributes Y; the attributes Y are a subset of $R(r)$ and are called *arithmetic attributes.*

If $X \in R(r)$, then the old attribute is replaced by the new attribute X with values specified by F and $R(\text{result}) = R(r)$. If $X \notin R(r)$, then the relation scheme of the result is extended with X and this column is filled for all tuples with the values specified by F.

Rename

The *rename* operator changes names of attributes. It is denoted by RN $(r, A_1 \leftarrow A, B_1 \leftarrow B, \cdots)$. We only consider a rename that introduces an attribute name that does not already exist. A rename RN $(r, B \leftarrow A)$, where B is an existing attribute, is equivalent to a calculate CL $(r, B \leftarrow A)$ followed by a remove column that eliminates A. It is introduced and treated as such in the paper.

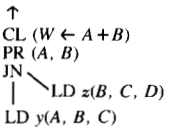


Figure 1 Tree representation of a relational expression.

Implied project

When the requirements for $R(r)$ and $R(s)$ in union, intersection, difference, and exclusion are not met, *implied projects* are applied that remove a minimum of attributes from r and s such that these requirements become satisfied.

Relational expressions

Relational expressions can be viewed as a sequence of relational assignments of the form $x_i \leftarrow$ relational operation. A sequence of assignments $x_1, x_2, \cdots, x_n$ can be reduced to one assignment x_n when we eliminate $x_1, x_2, \cdots, x_{n-1}$. For example,

$x_1 \leftarrow y$ JN z,

$x_2 \leftarrow$ PR $(x_1, \{A, B\})$,

$x_3 \leftarrow$ CL $(x_2, W \leftarrow A + B)$,

with $R(y) = \{A, B, C\}$ and $R(z) = \{B, C, D\}$ gives $R(x_1) = \{A, B, C, D\}$, $R(x_2) = \{A, B\}$, and $R(x_3) = \{A, B, W\}$. After elimination of x_1 and x_2, we obtain

$x_3 \leftarrow$ CL (PR (y JN $z, \{A, B\}$), $W \leftarrow A + B$).

Reversed Polish notation

A parenthesis-free notation for the relational expression can be obtained by introducing a conceptual stack upon which the relational operators operate. Relations are accessed by the operator *load*, written LD r. This operator conceptually loads a table on top of the stack. A monadic operator operates on the top of the stack; its result replaces the top of the stack. A dyadic operator operates on the two top tables of the stack, removes these tables from the stack, and places the result on top of the stack.

Tree representation of relational expressions

Relational expressions can be represented by binary trees. The nodes of such a tree correspond with the relational operators. The root, which delivers the result, is at the top of the tree; the leaf nodes, which load the operands, are at the bottom of the tree. The example given above appears now as shown in Fig. 1.

The attributes that are present in the various tables can be represented by a binary vector whose elements are true (represented by 1) when an attribute is present and false (0) otherwise. When the reversed Polish expression is written

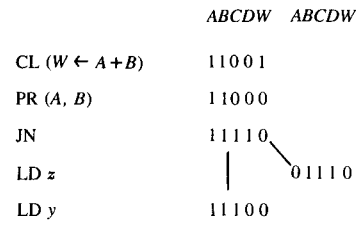


Figure 2 Tree representation of a relational expression showing attributes.

from bottom to top using an indentation for the attributes, it matches the tree structure. The example of this paragraph then is as shown in Fig. 2.

3. Evaluation of relational expressions

To demonstrate the optimization method we consider an actual implementation used for the evaluation of relational expressions, even though the method is not restricted to this implementation.

- *Row-by-row treatment*

In this implementation each node of the tree representation delivers its table row by row to the node above it when requested to do so. The first request originates at the root of the tree, the top node. When a row reaches the top node, a request for the next row is issued. Searching continues until the tree is exhausted.

A node corresponding to a monadic operator asks its lower level for the next row. As it receives a row, it determines whether this row fulfills the necessary criteria and, if so, transforms the row as required and submits it to its upper level. If the row does not fulfill the criteria, the node repeatedly asks for the next row until either a satisfactory row is obtained or the stream is exhausted.

A node corresponding to some dyadic operator, such as a join, asks for a row from the left subtree and a matching row from the right subtree. Depending on the kind of node, these rows are tested, combined, and sent upwards. For other dyadic nodes, such as a union, no matching occurs; rows are requested from the left subtree until it is exhausted; then, rows are similarly requested from the right subtree.

- *Access methods*

The relations in general reside in disk storage. The cost of accessing these relations can be expressed as the number of accesses to disk storage, called *I/O*'s.

Index

An index on one or more attributes may be used to retrieve a row from a table. We assume that for each stored table an index upon its key is available. If an index is not available on the desired attributes, we may decide to make one. Making an index has a certain cost which must be taken into account in the optimization process. The index is consulted to test for the presence of a particular row, as for a difference, intersect, or join. Depending upon the size of the index, this index inspection may involve one or more I/O's. If the particular row is not present, no further I/O is necessary. If it is present, extra I/O's may be necessary to fetch the particular row, as for the join.

Sort

Instead of using an index the participating tables may also be sorted on common attributes. In particular, in the absence of a suitable index upon an intermediate result, a sort-merge operation might be an attractive alternative. The optimization method can be used equally well to minimize the cost of sorts instead of the cost of index building, or even to decide which of these two methods is most advantageous. For the sake of simplicity, however, we always consider in this paper the use (and building) of an index rather than the equivalent sort operations.

Sequential scan

When the rows of a table can be used in the order in which they are stored, the table can be accessed sequentially. Such an access usually has lower cost than accessing the rows of the table via an index.

- *Cost*

The actual disk access time depends heavily on the available equipment, the general access methods used, and the implementation thereof. It is beyond the scope of this paper to treat this subject in detail. We assume here a highly simplified access-time computation and combine it through suitable parameters with an equally simplified processing-time computation to derive an overall cost figure. This computation, however, is independent of the optimization method and can easily be improved if more exact formulas for estimating time are available and are considered worthwhile.

In the current model the direct retrieval of a row of a table via an index to the table is considered to cost one I/O for each row. This cost is incurred for each matching row of the right operand of a join. The sequential retrieval of a table, in contrast, is considered to yield several useful rows per I/O and has a correspondingly lower cost per row. Sequential retrieval is normally used for a table that is accessed as the left operand of a dyadic operation, or as the single operand of a monadic operation, but also for the right operand of a quad or union.

The inspection of the index of a table to determine the presence or absence of a row without actually fetching that row is even lower in cost. This cost is used for the intersection and the difference.

The lowest cost is attributed to the processing time required by a select and calculate. In the model this cost is actually taken as zero, but as stated, it can easily be adjusted to another value.

The cost of making an index is higher than all these costs. As stated, we assume that this cost occurs when a table must be accessed with a key for which no index is available. In the given implementation duplicates are removed as an index is made. Therefore, the cost involved in duplicate removal is taken to be the same as that for making an index. The union and remove column operations are assumed not to involve any cost; the cost of the duplicate removal that is caused by them is separately accounted for.

- *Size*

The size of a relation is used as a parameter in reducing the cost of an expression. We assume that this size is known for stored tables. The relative size of intermediate results is obtained with a stochastic model.

Stochastic model

The stochastic model postulates for each attribute a set of occurring values of its domain. The size of this value set is assumed to be known. From the sizes of the tables that are input to an operation and from the size of the value set of the common attributes, the size of the resulting table can be obtained by computing its mathematical expectation. In reference [10] the size calculation is considered in greater detail.

Select

The reduction of a table as a result of a select operation depends upon the nature of the select expression. We assume that the reduction rate of this expression is known [11] and available to the optimizer as a parameter, called FR (fraction). Observe that the select expression uses only constants and the attributes of the current table. Since the values of FR are only used relative to each other, the absolute value of FR need not be known exactly.

The optimization of the select expression is beyond the scope of this paper. We note, however, that the expression of the select operator can be broken up into Boolean factors separated by AND operators [10, 12]. These factors each give rise to a new select operator which can be treated independently in the optimization process. We assume that this decomposition of selects precedes the optimization under discussion.

4. Transformation to set operators

- *Problem statement*

For a given relational expression the optimization process should deliver an expression with the same net effect, but with a minimal cost. A common strategy is to reduce the size of intermediate tables in an equivalent relational expression by interchanging adjacent operations. Existing systems have used this idea by pushing select operators towards the leaves of the expression tree [4–8]. Another general approach is to recognize common subexpressions and evaluate them only once [13, 14].

Our method transforms the relational expression into a suitable equivalent expression of set operations using intersection, union, and difference. The set expression is then transformed into a Boolean expression consisting of AND, OR, and NOT operations. This expression is optimized using logical minimization methods, but taking into account the constraints and the costs of the corresponding relational expressions. The logical transformation removes redundancy in the Boolean expression and copes with common subexpressions.

- *Universal relations*

Each relation can be treated as a set of tuples. The tuples of the various relations that appear in a relational expression, however, are generally not part of the same universe, since their elements belong to different attributes. Therefore, the first step of our method conceptually transforms the participating relations to relations whose tuples are members of a common universe, called *universal relations.*

For a relational expression with relations $r_1, r_2, \cdots, r_k$ we construct a universal relation ur with attributes $R(r_1) \cup R(r_2) \cup \cdots \cup R(r_k)$. The universal relation thus contains each existing attribute just once. The relation ur is filled with all occurring values as follows. Let $R(ur)$ consist of the attributes $A_1, A_2, \cdots, A_n$. For each attribute A_i of ur, we form a relation rr_i with the single attribute A_i such that $rr_i = \text{PR}\ (r_1, A_i)\ \text{UN PR}\ (r_2, A_i)\ \text{UN} \cdots \text{UN PR}\ (r_k, A_i)$. If A_i is not an attribute of $R(r_j)$, then the corresponding project does not participate in this union. The universal relation is now defined as $ur = rr_1\ \text{QD}\ rr_2\ \text{QD} \cdots \text{QD}\ rr_n$.

In other words, for every attribute A_i occurring in the relational expression, we place all occurring values of attribute A_i of any relation r in a one-column relation rr_i. The universal relation ur is formed by the Cartesian product of all the relations rr_i.

Having formed the universal relation, we replace every relation r_i by $r_i' = r_i\ \text{QD PR}\ (ur, R(ur) - R(r_i))$. Thus we project out of the universal relation ur those columns that are not present in r_i and form the Cartesian product of this table with r_i.

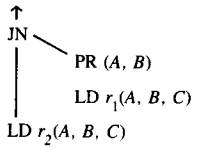


Figure 3 Relational expression with a project below a join.

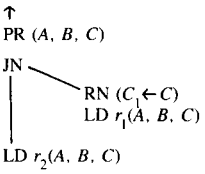


Figure 4 Relational expression with project moved to the top of the tree.

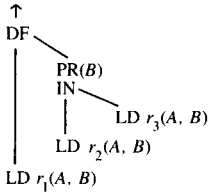


Figure 5 Relational expression with a project below a difference.

We can now verify that the following relationships hold:

$r_i' \subseteq ur$,

$r_i \subseteq \text{PR}\ (ur, R(r_i))$,

$r_i = \text{PR}\ (r_i', R(r_i))$,

$r_i\ \text{JN}\ r_j = \text{PR}\ ((r_i'\ \text{IN}\ r_j'), R(r_i) \cup R(r_j))$,

$r_i\ \text{QD}\ r_j = \text{PR}\ ((r_i'\ \text{IN}\ r_j'), R(r_i) \cup R(r_j))$,

$r_i\ \text{IN}\ r_j = \text{PR}\ ((r_i'\ \text{IN}\ r_j'), R(r_i))$,

$r_i\ \text{DF}\ r_j = \text{PR}\ ((r_i'\ \text{DF}\ r_j'), R(r_i))$,

$r_i\ \text{UN}\ r_j = \text{PR}\ ((r_i'\ \text{UN}\ r_j'), R(r_i))$.

The introduction of universal relations permits transformation of joins and quads into intersections and makes it possible subsequently to treat intersection, difference, and union as set operations.

Of the remaining relational operators, the select and calculate are transformed such that they also can be treated as set operators; the project and rename are moved to the periphery of the expression, where they affect only the output of the expression or its input. (Thus they need not be transformed to set operators.)

Project

As a rule we move all project operators to the top of the tree. There they can be combined and applied to the universal relation, in what we call a *universal project*.

Columns which are removed by the project operator and would otherwise participate in a join or intersection must be renamed before the project can be moved to the top of the tree. Take, for example, PR (r, A, B) with $R(r) = \{A, B, C\}$. We rename attribute C of $R(r)$ to C_1. Thus, if C participates in a join that follows the project, there is no conflict. After this rename, we can move the project to the top of the tree. This case is illustrated in Fig. 3, where tables r_1 and r_2 have attributes $\{A, B, C\}$.

The tree of Fig. 3 can now be replaced by the tree of Fig. 4, with the project appearing at the top. The combination RN and LD can be treated as a new table, with attributes A, B, C_1. This prevents column C from being used as a join column.

In case the right operand of a difference contains a project, we again rename the attribute that is removed by the project. The left operand must also be extended with this attribute such that the difference applies to universal relations. In contrast to the intersection, however, the extension of the left operand is not with the full domain of the removed attribute, but with such a subset of this domain that the net effect of the difference is not altered. This requirement is taken into account in the subsequent minimization.

In the example of Fig. 5, attribute A is projected away. We rename A to A_1 in the right subtree and expand the left subtree with an attribute A_1 with such values that the new expression is equivalent to the original expression. The project can now be moved above the difference.

Select

The select can be transformed into an intersection with a so-called *select table*. For select SL (r, F) we obtain the select table, rselect, by selecting out of relation r those tuples for which F becomes true; hence rselect = SL (r, F), and SL (r, F) can be replaced by r IN rselect. In the examples this substitution is assumed to be made; hence a select is shown as a select table followed by an intersection.

Calculate

The calculate is transformed into a join with a so-called *calculate table*, rcalc. As a first step we introduce project and rename operators to avoid conflicts between the attribute generated by the calculate and existing attributes. There are three types of calculates illustrated by the examples in Figs. 6, 7, and 8. In each case the expression at right is obtained from that at left.

Figure 6 illustrates a calculate that changes a column using its old value. We first rename the column B, which is to be changed, to B_1. Then we use B_1 in the calculate and subsequently remove it by a project. In general, with X in $R(r)$ and X used by F, CL $(r, X \leftarrow F(X, Y))$ is replaced by PR (CL (RN $(r, X_1 \leftarrow X)$, $X \leftarrow F(X_1, Y)$), $R(r)$), where Y represents all attributes that are not changed. If a calculate produces a new value for an existing column without using that column, the transformation is basically the same. This case is shown in Fig. 7. With X in $R(r)$ but not used by F, CL $(r, X \leftarrow F(Y))$ is replaced by PR (CL (RN $(r, X_1 \leftarrow X)$, $X \leftarrow F(Y)$), $R(r)$). When, however, a new attribute is generated, no change is required (Fig. 8). With X not in $R(r)$, CL $(r, X \leftarrow F(Y))$ remains unchanged.

The calculate operators now have the property that they calculate only new columns.

We next replace a calculate CL $(r, X \leftarrow F(X_1, Y))$ by r JN rcalc with rcalc = PR (CL $(r, X \leftarrow F(X_1, Y))$, X, X_1, Y). In case X_1 and Y are empty, F is a constant and the join becomes a quad. These substitutions are again assumed to have been made in the examples shown in this paper.

The artificially introduced select and calculate tables are called *special tables*. In contrast, the tables of the original relational expression are called *regular tables*.

Rename

As stated earlier, the renames that are part of the original expression are treated as calculates. But the renames that are generated by projects and calculates are pushed down to the leaf nodes of the tree. They are then combined with the stored tables. These tables thus obtain new attributes and become distinct from the same table in which this transformation is not performed. When a table is used at several places in a relational expression, the corresponding set expression may have different variables at those places because of this renaming.

Load

A project that applies directly to the load of a regular table is combined with that table, thus introducing a new table, instead of moving the project to the top of the tree. When the removed attributes are part of the key of the table, however, the project is always moved to the top of the tree.

Construction of universal tables

The relational expression can now be rewritten in terms of universal relations. With the relational expression written in reversed Polish notation, the operators are processed as they are encountered:

1. Calculate is replaced by a join with a calculate table, and may produce a rename and a project.
2. Project may produce a rename.

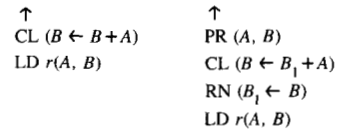


Figure 6 Transformation of a calculate that uses and changes a column.

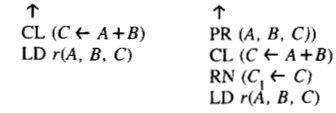


Figure 7 Transformation of a calculate that gives a column a new value.

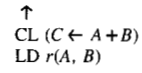


Figure 8 A calculate that generates a new column.

3. Rename produces new attributes and new tables.
4. Select is replaced by an intersection with a select table.
5. Joins and quads are replaced by intersections.
6. Unions, differences, and intersections remain unchanged.

The process described above leads to an expression consisting exclusively of unions, differences, and intersections, which can be interpreted as an expression of set operations.

Preservation of information

The universal project that precedes the relational expression is preserved as such. The relations involved in the expression are the extended regular relations and the extended special select and calculate tables. For the regular relations we remember their original attributes and keys. For the special tables we keep a record of the arithmetic and free attributes that are required for the operation from which they were derived. The nature of a special table, such as a select or calculate, is also recorded, including the select or calculate expression.

The preservation of information ensures that no information is lost that is needed for optimization and back transformation.

Logical optimization

• *Transformation to logical operations*

The expression of set operations is replaced by a logical expression by substituting for the union an OR, for the difference an AND-NOT, and for the intersection an AND.

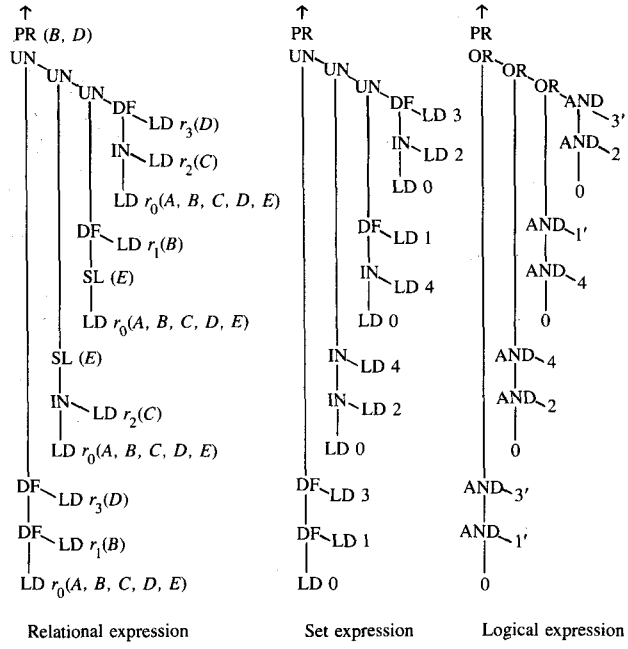


Figure 9 Example of the transformation of a relational expression to a logical expression.

The logical expression operates on a set of variables associated with the regular and special tables.

A logical expression is a logical function whose variables can take either the value true (1) or false (0). A logical function can be specified by tabulating the function as a so-called *truth table*. Thus the relational expression of Fig. 9 yields a logical expression. By tabulating the output of this expression for all 32 possible input values, the truth table of Fig. 10 is obtained. The names of the input variables, derived from the relations r_k, are chosen in such a way that a binary number can be derived from them. The suffix k corresponds to a bit in the binary number, where suffix 0 corresponds with the most significant bit. In Fig. 10 the binary number is represented by 5 bits. The truth table contains all possible 5-bit input values $0 \cdots 31$.

The truth table can be condensed by giving only the rows for which the function value is true. By giving the decimal equivalent of the binary codes, we can write the truth table even more compactly. Thus the truth table of Fig. 10 becomes 16 17 19 20 21 23 28 29 31. For example, 28, which in binary is 11100, means $r_0 \wedge r_1 \wedge r_2 \wedge (\sim r_3) \wedge (\sim r_4)$.

When this specification is given in a Karnaugh diagram [15], as in Fig. 11, we can visually observe the binary encoding. The Karnaugh diagram is akin to the Venn diagram. It displays the relation between the variables and codes, which we use in the minimization.

Observe that Figs. 10 and 11 no longer contain the particular structure of Fig. 9, yet they are its exact logical equivalent. Therefore, the truth table and the associated information about its variables is a general and neutral starting point for the minimization process.

- *Logical expression optimization*

The goal of the logical expression optimization is to minimize the cost of executing the corresponding relational expressions, as defined in the section "Cost." The optimization of these logical expressions is akin to the minimization problem in digital switching theory [16]. We briefly mention the major concepts of this theory. For each function specification, we can always find a so-called *canonical form* which corresponds to the OR of several terms whose factors are separated by AND and AND-NOT. We next change the canonical form such that the number of terms is reduced as well as the number of factors in the terms. Subsequently, we verify whether these terms can be realized. If not, an inversion is applied resulting in a new set of terms. The realizable terms are then optimized by changing the order of the factors in the terms.

Canonical form

The minimization starts with the specification of the logical function as a truth table, as indicated in the section "Transformation to logical operations." The rows of the truth table that are true can be satisfied by a sequence of terms separated by OR operators. Each term comprises all variables separated by an AND or an AND-NOT operator. This is the *canonical form*, and the terms are called *canonical terms*. Each variable occurs once and only once as a factor in a canonical term.

The canonical form is represented by a series of terms such as $(0 \wedge 1' \wedge 2' \wedge 3' \wedge 4') \vee (0 \wedge 1' \wedge 2' \wedge 3' \wedge 4)$, where 1' means NOT 1, and the variables 0, 1, 2, 3, 4 are abbreviations of r_0, r_1, r_2, r_3, r_4, respectively, and correspond to relational tables. This example shows the first two terms, terms 16 and 17, of the function for the truth table of Fig. 10.

Constraints

Each term of the ultimate expression must obey the constraints that are inherent in the corresponding relational expression. Otherwise this expression is not realizable.

First, a factor preceded with a NOT is called a *negative factor*. Such a negative factor may not appear first in a term since we cannot manage the corresponding complement of a relation.

Second, for t_i AND NOT t_j the attributes $R(t_j)$ should be a subset of $R(t_i)$. Otherwise, the corresponding difference operator is not realizable.

Third, a factor corresponding to a special table is called a *special factor*. Such a special factor must be preceded by other factors such that the arithmetic attributes of the special tables are a subset of the attributes provided by the preceding tables.

Prime implicants

The number of terms and the total number of factors that appear in all the terms can be reduced by finding the prime implicants of the given expression, as is standard practice for logical minimization. This reduction is based on the theorem $(x \wedge y) \vee (x \wedge y') = x$, which is based on the following postulates of Boolean algebra:

Distributive law: $(x \wedge y) \vee (x \wedge z) = x \wedge (y \vee z)$,

Complement law: $x \vee x' = 1$, with $1 =$ true,

Identity law: $x \wedge 1 = x$.

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

A Boolean function f implies a function g if for every v satisfying $f(v) =$ true it is also the case that $g(v) =$ 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 t_1 *subsumes* a term t_2 if all the variables of t_2 are contained in t_1. A *prime implicant* of a given logical function f is an implicant of f such that no other term subsumed by the prime implicant is an implicant of f.

In the example of Fig. 10 the prime implicants are

$0 \wedge 1' \wedge 3'$ (16 17 20 21),

$0 \wedge 1' \wedge 4$ (17 19 21 23),

$0 \wedge 2 \wedge 3'$ (20 21 28 29),

$0 \wedge 2 \wedge 4$ (21 23 29 31),

with in each case the function values shown between parentheses. In the Karnaugh diagram of Fig. 11 these prime implicants are indicated by an oval. The adjacency of terms is displayed by adjacency of position (perhaps across the boundary) in small diagrams.

We now replace the canonical solution of 9 terms and 45 factors with the prime implicant solution of 4 terms and 12 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 prime implicants is selected such that all function values are satisfied with a minimum number of terms.

r_0	r_1	r_2	r_3	r_4	Value
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	1
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	1
1	0	1	0	1	1
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	1
1	1	1	0	1	1
1	1	1	1	0	0
1	1	1	1	1	1

Figure 10 Truth table for the expression of Fig. 9.

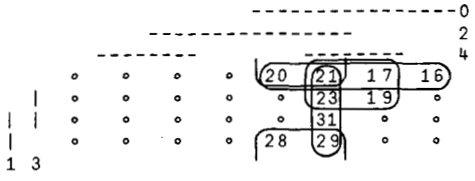


Figure 11 Karnaugh diagram of Fig. 10.

The algorithm to determine the prime terms is based on Quine [16]; that for finding the minimal cover is based on McCluskey [17]; both are used in the prototype, which is described in [10].

The prime terms may overlap. This means that they may have canonical terms in common. For example, both (16 17 20 21) and (17 19 21 23) satisfy 17 and 21.

Prime terms that overlap may create duplicates. Hence, after the corresponding union of subtrees, a duplicate removal is necessary.

Identities

The representation of a logical function with prime implicants eliminates any redundancies that may have been part of the relational expression from which the logical function is derived. The logical minimization, however, does not take

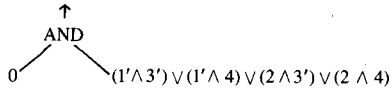


Figure 12 Common factor with right subtree.

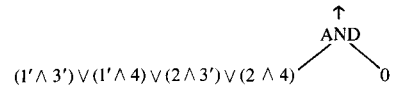


Figure 13 Common factor with left subtree.

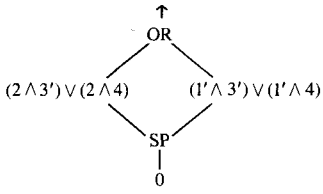


Figure 14 Use of the split operation.

into account the nature of the tables that are part of the corresponding relational expression. Some tables may permit further reduction of the logical expression.

Derived tables

Tables that are derived from others through a generated rename are called *derived tables*, and the corresponding factors are called *derived factors*. As an example, in the term $X1 \wedge Z2 \wedge Y1 \wedge X2 \wedge Y2$, the factors $X1$ and $Y1$ are derived from tables X and Y by renaming their attributes P and Q to P_1 and Q_1, whereas the factors $X2$, $Y2$, and $Z2$ are derived from X, Y, and Z by renaming only P to P_2.

Derived factors corresponding to derived tables in which the same attributes are renamed to the same new attributes, or are deleted, are called *compatible derived factors*. Thus, $X1$ and $Y1$ are compatible derived factors, but not $X1$ and $X2$.

A set of all derived factors within a term that are mutually compatible is called a *complete set of factors*. In the given example $X1$ and $Y1$ form a complete set of factors, but not $X2$ and $Y2$, since $Z2$ should be included as well.

One complete set of positive factors is considered a *compatible subset* of another complete set of positive factors if the originals of the second set subsume the originals of the first set and if the renamed attributes of the second set are a subset of the renamed attributes of the first set. For our example $X1 \wedge Y1$ is a compatible subset of $X2 \wedge Y2 \wedge Z2$ because $X \wedge Y \wedge Z$ subsumes $X \wedge Y$ and the renamed attribute P is a subset of the renamed attributes P and Q.

We can now state that if within one term there are two complete sets of factors, such that the first set is a compatible subset of the second set, then the first set can be removed.

Second, if within a term a set of factors is negated and that set is a complete set of factors that is a compatible subset of a complete set of factors in the remainder of the term, then the entire term can be deleted.

Third, if within a term two sets of factors are negated, and the first set is a complete set of factors that is a compatible subset of a complete set of factors within the second negated set of factors, then the second negated set of factors can be deleted.

Fourth, if a term contains one or more complete sets of factors such that each is a compatible subset of a complete set of factors in a second term and if the other factors of the first term also appear in the second term, then the second term can be deleted.

These rules are summarized by considering a complete set of factors $A1$ that is a compatible subset of another complete set of factors A. Then the following equalities hold:

$$A \text{ JN } A1 \Longleftrightarrow A1 \text{ JN } A \Longleftrightarrow A, \tag{1}$$

$$A \text{ DF } A1 \Longleftrightarrow \varnothing, \tag{2}$$

$$(B \text{ DF } A1) \text{ DF } A \Longleftrightarrow (B \text{ DF } A) \text{ DF } A1 \Longleftrightarrow B \text{ DF } A1, \tag{3}$$

$$(B \text{ JN } A) \text{ UN } (B \text{ JN } A1) \Longleftrightarrow B \text{ JN } A1. \tag{4}$$

In Eqs. (3) and (4) the factor B has the necessary attributes to satisfy the constraints of the expression.

• *Common factors*

The solution with prime implicants can be further improved by exploiting the occurrence of common factors. If some terms have one or more factors in common, we can combine these terms by isolating the common factors.

For the example of Fig. 9 all prime terms have the factor 0 in common. Hence we can write the expression as

$$0 \wedge (1' \wedge 3' \vee 1' \wedge 4 \vee 2 \wedge 3' \vee 2 \wedge 4).$$

There are several possible ways to proceed from this point.

First, f can be decomposed as shown in Fig. 12, with the common factor 0 as the left subtree and the compound factor as the right subtree.

We discard this solution because the relational equivalent requires an index for the right subtree. A second possibility has the common factor as the right subtree and the compound factor as the left subtree, as shown in Fig. 13.

If there is a usable index on 0, which is normally the case with a stored table, we could allow this possibility. The left subtree, however, contains a term with negative factors only. This term cannot be realized since each term must begin with a positive factor.

A third possibility is to ignore the common factor and build the terms separately. This has the disadvantage that the common factor is evaluated repeatedly.

To evaluate the common factor only once, we introduce in the next subsection an operation called "split" that has the property of sending the result of the common factor to two destinations.

Split operation

A *split* operator, written SP, has one input and two outputs. Split has the property of making its input available at both outputs with the input evaluated only once. Thus, the use of a split is always advantageous over the evaluation of separate terms. In the given example we save three sequential accesses of table 0.

To use the split we must partition the expression that follows the common term. This partitioning is performed by an algorithm that tries to get a maximum number of common factors in each of the two parts. We decompose the left subtree of Fig. 13 into two expressions, recognizing two terms with common factor 2 and two terms with common factor 1′, as shown in Fig. 14.

In the left subtree we factor 2, in the right subtree we factor 1′, which gives Fig. 15 as a result.

Next we use the split in the remaining expressions as shown in Fig. 16.

The relational expression that is equivalent to Fig. 16 is shown in Fig. 17, where we have replaced AND 2 by IN 2; similarly, AND 1′ is replaced by DF 1, OR by UN, etc.

Constraints of the split operation

The use of the split operation presupposes that the common term that serves as its input can be realized. This, in turn, requires that at least one positive factor be derived from a regular table in the common term. In the given example the common term is the positive factor 0, derived from r_0. Hence table r_0 can be accessed and its rows presented sequentially to the split operation. If the common term had been 0′ or a special table, this would have been impossible. In general the

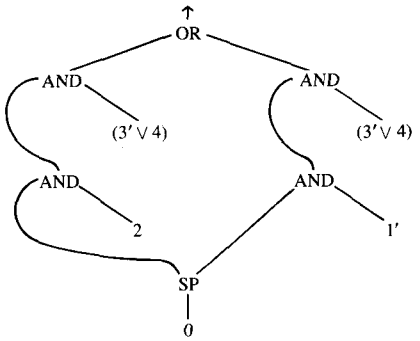


Figure 15 Further extraction of common terms.

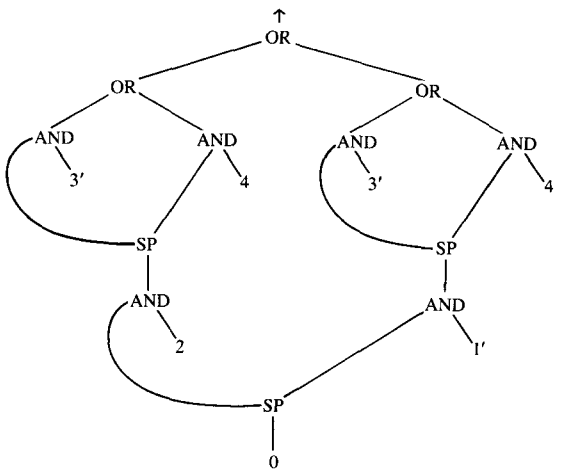


Figure 16 Repeated use of split operation.

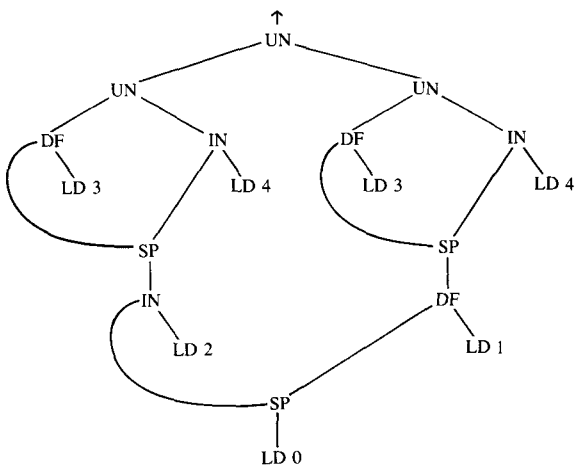


Figure 17 Relational equivalent of Fig. 16.

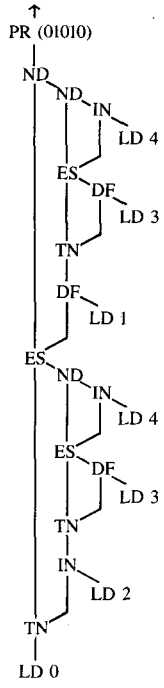


Figure 18 Solution of Fig. 17 written with stack operators.

common term can have several factors. As long as at least one of them is positive and regular, the split can be used. Common negative factors and special factors must be realizable; that is, all attributes of the negative factors and the arithmetic attributes of the special factors must be covered by the common positive factors. The nonrealizable factors are placed in the branches following the split. Once the split can be used, subsequent splits can always be realized, as illustrated in Fig. 16.

Semantics of the split operation

If the prime terms that are combined by a split operation overlap, the corresponding union may deliver duplicate rows. It is possible to avoid these duplicates, however, by a suitable choice of the semantics of the split and its corresponding union [10]. Because this union is a particular function, we denote it by ND (for "end") in contrast to the union UN, which may deliver duplicates.

Stack operators for the split function

We introduce the two dyadic operators TN and ES, which are abbreviations of THEN and ELSE, respectively, which operate on the top of the stack of tables. The operator TN conceptually duplicates the top of the stack. The operator ES reverses the two top locations of the stack. With the use of TN and ES we can write a solution in reversed Polish notation. Thus, the example of Fig. 17 can now be written as in Fig. 18. The row-by-row implementation of TN, ES, and ND avoids the string of the result of the common subexpression.

• *Nonoverlapping terms*

When there are no regular positive common factors, the split cannot be used and the solution must use a union combining two sets of prime implicant terms. Again a partition algorithm is used to divide the terms into two groups. This time the algorithm optimizes the occurrence of regular positive common factors in each of the two groups, such that for these groups the use of a split is favored.

In general the union that combines these two groups will meet duplicates in the rows that it receives from its two operands. Since a duplicate removal is a costly operation, we accept overlapping terms only if a duplicate removal is necessary anyway for other reasons.

When at least one of the groups can be realized without duplicates, we change the second set of terms so that it does not overlap the first set. The union now reduces to a disjoint union DN because nonoverlapping terms have no common tuples. As an example we consider the expression in Fig. 19. The specification function is given in the Karnaugh diagram in Fig. 20. We can derive two nonoverlapping terms by taking

$0 \qquad (2\ 3),$
$1 \wedge 0' \qquad (1).$

Therefore, an equivalent solution is

$0 \vee 1 \wedge 0'.$

The corresponding relational expression is shown in Fig. 21. It trades the duplicate removal of the overlapping union for the index inspection of a difference. A more elaborate application of this principle is found in Example 6, Section 8.

• *Negative terms*

We observed in the creation of universal tables that a project in the right subtree of a difference introduces a renamed attribute in the left subtree of the difference whose value set depends upon the right subtree. Such a value set cannot easily be realized. But it need not be realized if the renamed attributes of the right subtree are properly combined in the ultimate expression. For the logical terms this means that the attributes of a negative factor must be covered by the attributes of the positive factors. If not, the terms in which these attributes occur are inverted by applying De Morgan's theorem: $(\sim x) \vee (\sim y) = \sim(x \wedge y)$. This negative term is then combined with a common factor through a difference. Thus, the two terms obtained for the example of Fig. 5 are $1 \wedge 2'$ and $1 \wedge 3'$. In these terms the renamed attribute A_1 of the negative factors $2'$ and $3'$ is not present in the regular positive factor 1. Hence the expression is rewritten with De Morgan's theorem as $1 \wedge \sim(2 \wedge 3)$. Therefore, in this example the optimized result is the same as the original. In

general, however, some optimization within these constraints is still possible, as illustrated by Example 8, Section 8.

• *Decomposition algorithm*

The steps of the decomposition algorithm encountered so far are: Obtain the canonical form from the relational expression; derive the prime implicants; determine a minimal cover.

We adopt the following strategy for splitting a cover of more than one term:

IF a regular positive common factor has occurred in the past or occurs now
THEN
  IF all terms involve attributes that are not properly covered in some terms
  THEN determine the inverse of those terms and apply one or more differences to the inverted terms.
  ELSE apply a split operator and split the cover into two sets of prime implicants, the first of which contains all terms with improperly covered attributes; if the first set is empty, the split is such that the number of common factors is optimal.
  ENDIF
ELSE split the prime implicants into two parts such that the parts have an optimal number of regular positive common factors.
  IF the universal project does not spoil the key of the first part
  THEN specify the second part nonoverlapping with the first (this normally involves the derivation of a new set of prime implicants).
  ELSE place a DR (duplicate removal) above the union unless a DR is set subsequently, and retain the overlapping prime implicants of the second part.
  ENDIF
ENDIF

Repeat the algorithm until a single term is obtained. For a single term the DR is placed at the end of the term when the key is spoiled.

6. Term optimization

Ultimately the original expression is broken up into terms, separated by splits, differences, and unions. A term may be located at the periphery of the tree; it can also be situated between a TN and ES or an ES and ND. In the term there are positive factors and negative factors of regular or special tables. Factors may be interchanged taking the constraints into account. Also, the term should contain the necessary index generation and duplicate removal.

• *Term optimization criteria*

The term optimization uses the following heuristics, which reflect the cost criteria mentioned in the section on cost.

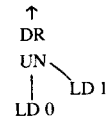


Figure 19 Overlapping union.

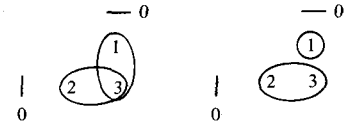


Figure 20 Karnaugh diagrams for overlapping and nonoverlapping unions.

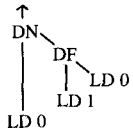


Figure 21 Nonoverlapping union.

First, the generation of an index is avoided where possible; second, an intersection is preferred over a join; third, a special table is preferred over a regular table; fourth, a join or difference with a table that gives a large reduction is placed earlier than one that gives a smaller reduction.

• *Nature and reduction of factors*

As is common for a hierarchy of criteria, they are applied in ascending order, starting with the weakest and ending with the strongest. Thus the terms are first placed in the ascending order of the fraction of the value set that their table contains (for the positive terms) or does not contain (for the negative terms). Next, the special tables are placed in front of the regular tables, leaving the order within each set unchanged.

As an illustration we refer to Example 5 of Section 8. This expression has the logical equivalent of just one term of eight positive factors $0 \wedge 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge 6 \wedge 7$. The factors 0, 1, 2, 3 are regular factors with a fraction of 1; the factors 4, 5, 6, 7 are special factors derived from selects with a fraction of 0.7, 0.9, 0.4, 0.5, respectively. After the sort, their order becomes 6 7 4 5 0 1 2 3.

• *Non-index groups*

The factors are now grouped. Starting with each factor, a group is formed of all factors that can be combined without

recourse to building an index. Hence regular positive factors should have their key attributes covered by the attributes of preceding factors, negative factors should have all attributes covered, and special positive factors should have their arithmetic attributes covered. Positive regular factors give coverage with all their attributes; positive special factors give coverage only with their free attributes.

When the term is preceded by the common factors of a split, the positive factors of the common term give a starting cover. Otherwise, there is no starting cover. When a positive factor is placed in a group, its attributes can be used in turn to cover other factors. Hence grouping is a converging iterative process.

For our example the nature of keys and arithmetic attributes shown in Section 8, Example 5, gives the eight groups 0, 1 4, 2 5, 3 6 7 5, 4 1, 5, 6 7 5 3, 7 6 5 3.

- *Meta-groups*

When all groups are obtained, those groups that are a proper subset of another group are eliminated. This leaves a set of groups such that each factor is in at least one group, and all groups start with a different factor. The groups are now combined into *meta-groups*; all groups within a meta-group contain the same set of factors. One member of a meta-group must be selected and the order of these selected groups must be established.

From the eight groups of our example the group containing table 5 is eliminated, since it is a proper subset of group 2 5. Next, four meta-groups with 1, 2, 1, and 3 members each are obtained: 0; 1 4, 4 1; 2 5; 3 6 7 5, 6 7 5 3, 7 6 5 3. Observe that the factor 5 appears in two meta-groups.

- *Order of groups*

We now establish all possible orderings of groups. First we determine all permutations of the meta-groups. Next we replace for each permutation every meta-group, in turn, by each of its members.

Thus, for the four meta-groups of our example, 24 meta-group permutations are derived, and by substituting groups for meta-groups, $24 \times 1 \times 2 \times 1 \times 3 = 144$ orders are obtained. These 144 orders to be investigated are far less than the 40 320 permutations of the eight constituting factors.

- *Valid group orders*

The number of group orders is reduced by first testing the validity of each order. Each group should start with a regular positive factor unless the first factor of that group is suitably covered by the attributes of the preceding groups or of a possible common term. In our example the number of orders is thus reduced from 144 to 24.

- *Maximum key join*

The valid group orders are further reduced by taking those orders that allow a maximum number of key joins between groups. The key join applies only to groups whose starting factor is covered by more than one preceding group. In the given example no key joins between groups are possible, so the number of orders remains unchanged.

- *Minimal join cost*

From the remaining orders the one with minimal join cost is finally selected. For each order the intersecting value sets are determined using the stochastic model and from this the increase or decrease in size that results from joining the groups. The sum of all intersection sizes then gives a relative cost figure.

For our example the order 2 5, 3 6 7 5, 0, 1 4 is selected, which has intersecting sizes of 2, 4, and 20, as shown in Example 5, with a relative cost figure of 26. Placing the group 1 4 first instead of last, for instance, would have given intersecting sizes of 30, 60, and 120 with a cost figure of 210.

- *Order within groups*

Now that the order of the groups has been established, the order of the factors within each group can be determined. Since the starting cover of the group is known, the factors whose attributes are covered are placed in the order established in the section "Nature and reduction of factors" to form ultimately intersections or differences. The remaining factors that have their key or arithmetic attributes covered, ultimately resulting in joins, are now investigated in turn. The least expensive solution is obtained using backtracking. After the placement of each join, intersections and differences may become possible again; the process is repeated until all factors are placed. A relative cost factor, derived from the stochastic model, is applied such that the optimum solution can be retained. In our example the order within the groups is not changed.

7. Transformation of logical to relational operations

Once a suitable form has been obtained for the logical analogon, its expression must be transformed back to a relational expression. Formally, two steps are involved. The first maps logical operators upon set operators, which is trivial. The second step maps the set operators upon the relational operators such that an expression in terms of the original tables, as well as the original select and calculate operators, is obtained.

- *Table transformation*

Since for each variable the originating table is known, a load of a variable can be changed to a load of a table. If the table

was created out of an original table by renaming one or more of its attributes, the derived table name is replaced by the original table name and rename operators are introduced as needed.

- *Remove column*

The project appears as the universal project at the top of the tree at the start of the reverse transformation. This project is decomposed into a remove column, RC, and a duplicate removal, DR. During the transformation the attributes of the upper and lower levels of the tree nodes are determined for each node. The remove column can pass through a union to the left as well as to the right subtree. Since all unions appear at the top of the tree (or a negative subtree), the universal project applies to each term.

The remove column may only be pushed through a join provided the common attributes of the join remain unchanged. In that case the remove column is pushed to the left or the right subtree. If, however, a remove column would omit common attributes, such a remove column would remain above the join. A new remove column that is composed of the union of the attributes of the old remove column and the common attributes can be pushed through the join.

Similar rules apply to the quad, intersection, and difference.

When two remove columns meet at a split, we distinguish two cases: If the remove columns are equal, they are united as one remove column below the split. If the remove columns are different, a remove column which omits the attributes that are absent in both remove columns is pushed through the split. At the left branch of the split a remove column remains which omits attributes that need omission in the left branch, but are required in the right branch. In the right branch a remove column is placed for the reversed situation.

Although these rules appear quite complex, they follow directly from the nature of the remove column and the nodes that are encountered. Furthermore, the term optimization provides a structure to the term, which makes the placement of the remove column nodes quite straightforward.

Figures 22 and 23 show the result of placing the remove columns for the example of Fig. 18. Figure 22 gives the starting situation. Next to the nodes the attributes of the tables and the attributes of the intermediate results are shown. On top of the tree there is the universal project. Since the project does not spoil the key of the result, the project can be replaced by a remove column in the final expression; since, furthermore, the unions are replaced by ND operations, there is no duplicate removal. Figure 23 shows the original expression and its optimized result using the reversed Polish notation described earlier and giving the details of the

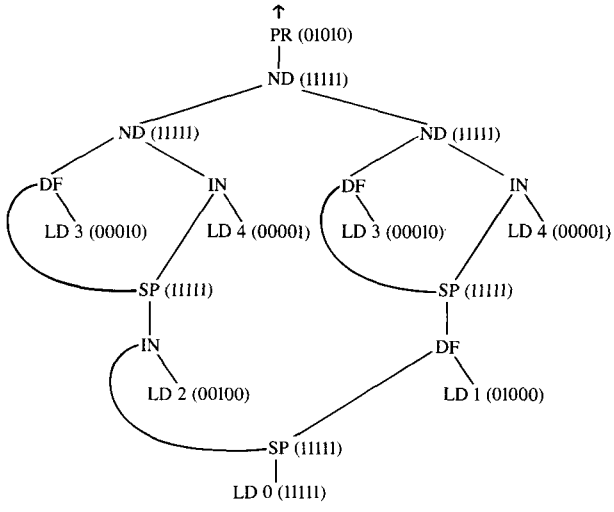


Figure 22 Tree with attributes and universal project.

examples of Section 8. Note that the prototype makes a different (but equivalent) choice in using the TN and ES branches.

- *Type of intersection*

As the remove columns are placed during the reverse transformation, some of the intersections are replaced by joins or quads in accordance with the definitions of Section 2.

- *Type of table*

The reverse transformation of LD selecttable IN transforms back to SL (F), where F is the select function associated with the select table. The reverse of LD selecttable DF is SL (NOT F). In our examples, however, we keep the form with the select table.

Similarly, the reverse transformation of LD calculatetable JN results in CL ($X \leftarrow F$), where F is the arithmetic expression of the calculate. The case of LD calculatetable IN is replaced by SL ($X = F$); LD calculatetable DF is replaced by SL ($X \neq F$).

- *Index generation*

In the term optimization we have accounted for the building of an index for a stored table that occurs as right operand of some join. At those places an index operator is placed, written IX (Y), where Y is the set of attributes for which the index should be built.

8. Examples

In this section the optimization method is illustrated by a few examples. In each case an original expression is given, using the notation of Section 2. Each figure gives the participating

```
SIZE COST          COLUMNS
 129 1310 PR   10 01010
 195      UN     11111
  89      UN           11111
  68      UN                 11111
  48    7 DF                       11111
   5      LD   3                          00010
  63   21 IN                       11111
   3      LD   2                          00100
 210   42 LD   0                    11111
  20    5 DF                 11111
   7      LD   1                       01000
  42      IN                 11111
   1      SL   4                    00001
 210   42 LD   0                    11111
  21      IN           11111
   1      SL   4                 00001
  63   21 IN             11111
   3      LD   2                 00100
 210   42 LD   0        11111
 106   14 DF     11111
   5      LD   3        00010
 137   21 DF     11111
   7      LD   1        01000
 210   42 LD   0 11111
SUM: 1567

TB ATTR  KEY    FR
 0 11111 01010  50
 1 01000 01000  34
 2 00100 00100  25
 3 00010 00010  20
 4 00001 00001  17
AT[VAL;]
20 20 10 21 5
F
16 17 19 20 21 23 28 29 31
```

```
SIZE COST          COLUMNS
 129      ND     01010
 108      ND           01010
 106   14 DF                 01010
   5      LD   3                       00010
 137      RC   10               01010
 137      ES              01010 01011
  20      RC   10               01010
  20      IN                    01011
   1      SL   4                       00001
 137      TN              01011 01011
 137   21 DF              01011
   7      LD   1                01000
 210      RC   11          01011
 210      ES     01010 01111
  53      ND              01010
  21      RC   10               01010
  21      IN                    01011
   1      SL   4                       00001
  63      ES              01010 01011
  48    7 DF                    01010
   5      LD   3                       00010
  63      RC   10               01010
  63      TN              01011 01011
  63      RC   11          01011
  63   21 IN               01111
   3      LD   2                00100
 210      TN     01111 01111
 210      RC   15 01111
 210   42 LD    0 11111
SUM:  105
```

Figure 23 Example of Fig. 22 after pushing of remove column.

tables (TB), their attributes (ATTR), and keys (KEY), as well as the fraction (FR) of the value set represented by the table. For the special tables, the arithmetic attributes are shown instead of the key. AT [VAL;] gives the size of the value set for each attribute.

Next, the analogous logical function value is presented in the form of a Karnaugh diagram, as explained in the section "Transformation to logical operations." In the diagram the terms of the optimized result are drawn as ovals. The function value is also given as *F* in linear form for the optimized result. The two function values should be the same, which is a necessary (but far from sufficient) check upon the output of the optimizer. Finally, the optimized result expression is displayed in the same manner as the input. The tables participating in the result are the same as for the input and are not repeated.

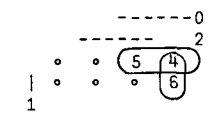
To the left of the original and the optimized expression, size and cost entries are shown at each point in the execution of the expression. This *size* is computed by simulation independent of the optimization. The simulator checks that the results of the original expression and the optimized expression are the same. The *cost* estimate is derived from the size estimate. Indexed access is given a relative cost value of 10, sequential access 0.2, and index inspection 0.1. Index generation and duplicate removal involve indexed access and index inspection. Fractional costs are rounded up to integers. The value *sum* gives the total cost. The sizes of tables and value sets are kept small to reduce simulation time; the cost values are relative anyway.

- *Example 1*

This example (Fig. 24) illustrates the use of the split operation (TN, ES, ND); the elimination of an index generation (IX); the change of a calculate (CL) into a select (SL); the implied remove column of a difference (DF); the use of one sequential scan (LD 0) instead of two (LD 0 and LD 1); the "pushing down" of the remove column (RC); changing the project into a remove column (RC), and duplicate removal (DR).

Placing a remove column early in the expression is an advantage to the implementation, but this is not shown in the cost figures. As mentioned in the section "Cost," a sort could be used instead of the index generation. The load of table 1 in the result has no cost, since it is a conceptual load; the index of table 1 is interrogated by the difference, with a cost of 4.

```
 SIZE  COST          COLUMNS
   21   214 PR   2 0010
   39     4 DF     0111
   13   132 IX   3        1011
   13       JN             1011
   17       CL   2                  0011
   13     3 LD   1        1010
   40     8 LD   0 0111
 SUM:   361

TB ATTR KEY    FR
 0 0111 0100   50
 1 1010 0010   60
 2 0011 0010   80

AT[VAL;]
10 80 21 10
```



```
F
4 5 6

 SIZE  COST          COLUMNS
   21   214 DR     0010
   37       ND     0010
   34     4 DF             0010
   13       RC   2                  0010
   13       LD   1                  1010
   40       RC   2        0010
   40       ES     0010 0011
   37       RC   2        0010
   37       DF             0011
   17       SL   2                  0011
   40       TN     0011 0011
   40       RC   3 0011
   40     8 LD   0 0111
 SUM:   226
```

Figure 24 Example 1.

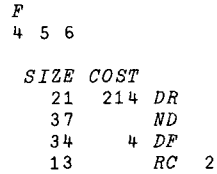
The join in the original has no cost, since it applies to a calculate table; the two operations should be taken as an entity that represents a calculate operation. Similarly in the result the difference that applies to a select table is without cost; the difference and the load of the select table form a select operation.

Several of these features reappear in later examples; we do not mention them each time.

- *Example 2*

This example (Fig. 25) changes a rather involved and redundant expression into a simpler nonredundant one. The duplicate removal implied by the final project is not necessary in the result. The algorithm recognizes that the two terms of the expression are not overlapping, and hence uses the nonoverlapping union (DN). In the result the difference with table 3 appears early in the first term and late in the second term, since table 3 is larger than table 1 and smaller than table 0. The split cannot be used in this example, since the only common factor is the difference with table 3.

```
 SIZE  COST          COLUMNS
   25   253 PR   3 011
   30     6 DF     111
   51   516 IX   7      111
   58       UN           111
   18   184 JN                 111
   50       LD   0                    111
   34     7 LD   1              110
   40     8 LD   3      111
   55       UN     111
   30     5 DF           111
   40       LD   3              111
   50    10 LD   0      111
   25     5 LD   2 111
 SUM:   994

TB ATT KEY   FR
 0 111 010   50
 1 110 010   34
 2 111 010   25
 3 111 010   40

AT[VAL;]
20 100 50
```



```
F
2 6 8 10

 SIZE  COST          COLUMNS
   25       DN     011
    6       RC   3      011
    6     2 DF           111
   40       LD   3              111
   12     3 DF           111
   50       LD   0              111
   25     5 LD   2      111
   19       RC   3 011
   19     3 DF     111
   34       LD   1      110
   30     5 DF     111
   40       LD   3      111
   50    10 LD   0 111
 SUM:    28
```

Figure 25 Example 2.

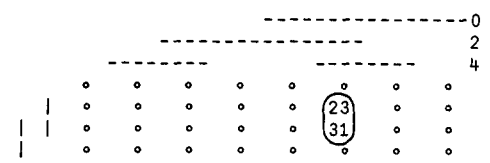
- *Example 3*

This example (Fig. 26) concerns a single term in which the order of the factors is changed to avoid index generation. This change in order, however, requires a rename so as to preserve the attributes over which the join must be made. The select, which appears in one of the branches of the original structure, now appears low in the main stem. Table 1 is not used.

- *Example 4*

This example (Fig. 27) illustrates the replacement of an exclusion (XC) by intersections (IN) and differences (DF). Observe the order of applying tables 1 and 2 in the two branches of the split, which depends on whether an intersec-

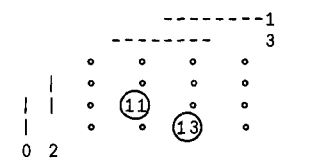
```
SIZE COST          COLUMNS
   1   11 JN      1011
  10  101 IX   1         1011
  10       IN             1011
   1       SL   3                   1000
  68   14 LD   4         1011
   9       RC   1 0001
   9   92 JN      0111
  15  152 IX   2         0111
  15    3 LD   2         0111
  15       RC   2 0010
  15    3 LD   0 0110
SUM:  376

TB ATTR KEY   FR
 0 0110 0010  50
 1 0010 0010  33
 2 0111 0001  25
 3 1000 1000  14
 4 1011 1001  16

AT[VAL;]
7 10 30 60
```




```
F
23 31

SIZE COST          COLUMNS
   1       RC   11 01011
   1    1 IN       11011
  15       RC   16         10000
  15       RN   18         10100
  15       LD    0         00110
   1   11 JN       11011
  15       RC   17         10001
  15       RN   18         10101
  15       LD    2         00111
  10       IN       01011
   1       SL    3         01000
  68   14 LD    4 01011
SUM:   26
```

Figure 26 Example 3.

```
SIZE COST          COLUMNS
   1       RC   1 01
   1       IN     11
   5       SL   3     10
  23    5 IN     11
  48  485 IX   1      01
  48    7 XC           01
  30    6 LD   2          01
  40    8 LD   1      01
  50   10 LD   0 11
SUM:  521

TB AT KE  FR
 0 11 01  50
 1 01 01  40
 2 01 01  30
 3 10 10  10

AT[VAL;]
50 100
```




```
F
11 13

SIZE COST          COLUMNS
   1       ND     01
       1 DF          01
  40       LD   1          01
   2    1 IN           01
  30       LD   2          01
   5       ES     01 01
   1    1 DF           01
  30       LD   2          01
   3    1 IN           01
  40       LD   1          01
   5       TN     01 01
   5       RC   1 01
   5       IN     11
   5       SL   3     10
  50   10 LD   0 11
SUM:   14
```

Figure 27 Example 4.

tion or a difference is used. The select (SL 3) moves from the end of the tree to its starting node. One branch of the split turns out to give an empty result.

- *Example 5*

This example (Fig. 28) is intended to show the placement of select operations. Originally four selects are placed at the end of a tree of quads (QD). The order is changed to replace one quad with a join and to allow the selects to appear early in the tree. One select (SL 7) changes into a calculate (CL 7). This is possible because the arithmetic attributes of this select indicate that attribute 3 can be derived from attribute 2. Select 4 is applied to the right branch of a quad. Select 5 is applied to the left and right of a join, thus reducing index generation cost. Duplicates are removed in the right subtree of the quad. The original expression generates many duplicates, which are removed in the project. Because of the small table sizes, the quads are relatively inexpensive. The Karnaugh diagram is not shown—with eight variables it would become rather large, and would moreover display only one function value.

- *Example 6*

In this example (Fig. 29) a duplicate removal caused by several unions is eliminated. Table 0 is sequentially scanned and used unaltered; this corresponds to function values 16–31 of the Karnaugh diagram. Next, table 1 is scanned, but its overlap with table 0 (function values 24–31) is eliminated through a difference with 0. Finally, function values 5 6 7 are obtained by a split preceded by the common

```
SIZE COST          COLUMNS
  10       IN      001111
   3       SL   7          001100
  60       IN      001111
   2       SL   6          001000
 150       IN      001111
   3       SL   5          000100
 150       IN      001111
   5       SL   4          000010
 210 2184  PR  15 001111
 840  336  QD      111111
  30   12  QD               101100
   5       LD   3                    001000
   6       LD   2          100100
  28   12  QD      010011
   7       LD   1          000010
   4    1  LD   0 010001
SUM: 2545

TB ATTR    KEY      FR
 0 010001  010000   100
 1 000010  000010   100
 2 100100  100000   100
 3 001000  001000   100
 4 000010  000010    70
 5 000100  000100    90
 6 001000  001000    40
 7 001100  001000    50

AT[VAL;]
6 4 5 3 7 2

F
255
```

```
SIZE COST          COLUMNS
  10  102  DR      001111
  20    8  QD      001111
   5       IN              000010
   5       SL   4                    000010
   7       LD   1          000010
   4    2  QD      001101
   2   21  DR              000001
   4       RC   1          000001
   4       LD   0          010001
   2   21  JN      001100
   1   11  IX   4          001100
   1       IN              001100
   3       SL   5                    000100
   1       JN              001100
   3       CL   7                    001100
   2       IN              001000
   2       SL   6                    001000
   5    1  LD   3          001000
   6       IN      000100
   3       SL   5          000100
   6       RC   4 000100
   6    2  LD   2 100100
SUM:  168
```

Figure 28 Example 5.

factors 2 difference 0, difference 1. The example has many implied projects caused by the unions and the intersection; they result in the remove columns that appear throughout the result.

- *Example 7*

This example (Fig. 30) illustrates the implied project of a calculate. As a small example it also shows how a join is replaced by an intersection and how the intersecting columns of joins are preserved with remove columns when the order of the factors changes.

- *Example 8*

This last example (Fig. 31) illustrates the recognition of negative terms that are otherwise not realizable. A reduction in cost is obtained by combining two such terms prior to the index generation.

- *Prototype of the optimizer*

Since the proposed method is intended to be general, a large variety of cases should be considered. The use of a prototype is almost indispensable in such a situation. Also, the many features and interactions that must be considered require the accurate description provided by the model.

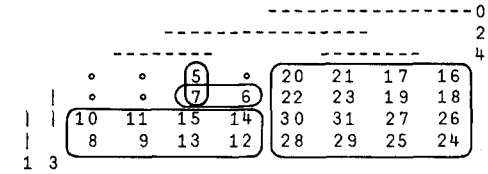
The examples of this section are samples of the cases that have been tested with the prototype of the proposed optimization method. The prototype gives a precise and complete description of the method concerned. As such it contains the essential algorithms. Because it is an executable description, the method can be demonstrated and tested for accuracy, consistency, and effectiveness. The prototype, however, is an architectural description and is not concerned with implementation matters, such as program performance, memory allocation, or data representation.

A prototype constitutes an important milestone in the management of a design. It ensures the correctness of a major part of the design and allows review and feedback prior to the implementation effort.

For the proposed method the prototype was written in APLDL [18], which is standard APL to which the regular control structures IF THEN ELSE, WHILE, CASE, REPEAT UNTIL, and FOR are added. This enhancement does not affect the APL interpreter, hence it is generally applicable; it improves legibility and facilitates the use of the prototype as a specification for implementation in a different language.

APL encourages the use of many short functions, each with a specific task. This language feature, combined with the control structures of APLDL, give a clear design structure. In Reference [10] the top levels of the model are shown

```
 SIZE  COST          COLUMNS
   81   822  DR      0001000
  120        UN      0001000
   60        UN              0011000
   10     4  IN                      0011101
   28   283  IX   56                         0111100
   30        UN                              0111100
    5     1  LD    4                                    0111110
   25     5  LD    3                         1111100
   40     8  LD    2                 0011101
   50    10  LD    1         0011010
   60    12  LD    0 0001111
 SUM:  1145

TB ATTR     KEY       FR
 0 0001111  0001000   60
 1 0011010  0001000   50
 2 0011101  0001000   40
 3 1111100  0001000   25
 4 0111110  0001000    5

AT[VAL;]
2  5  50  100  20  25  100
```




```
F
5  6  7  8  9  10  11  12  13  14  15

16  17  18  19  20  21  22  23  24  25

26  27  28  29  30  31
```

```
 SIZE  COST          COLUMNS
   81        DN      0001000
   21        DN              0001000
    2        ND                      0001000
             RC   8                             0001000
           1 IN                                0011100
    5        RC   28                                     0011100
    5        LD   4                                      0111110
    7        ES                      0001000 0011100
    2        RC   8                             0001000
    2     1  IN                                0011100
   25        RC   28                                     0011100
   25        LD   3                                      1111100
    7        TN                      0011100 0011100
    7     2  DF                      0011100
   50        RC   24                            0011000
   50        LD   1                             0011010
   15     4  DF                      0011100
   60        RC   8                             0001000
   60        LD   0                             0001111
   40        RC   28                 0011100
   40     8  LD   2                  0011101
   19     5  DF              0001000
   60        RC   8                  0001000
   60        LD   0                  0001111
   50        RC   8          0001000
   50    10  LD   1          0011010
   60        RC   8  0001000
   60    12  LD   0  0001111
 SUM:    43
```

Figure 29 Example 6.

such that the reader can obtain a more precise understanding of the algorithms employed.

9. Evaluation

• *General characteristics*

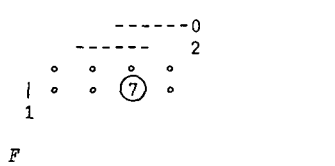
The method as described proves to be quite powerful. It is able to cope with a great diversity of items to be optimized, such as common subexpressions, empty tables, redundancy, the assurance of nonoverlapping intermediate results, the use of the split operation, the order of operations, and the minimization of index generation (or corresponding sort operations). It shows that it is possible to separate these many items using simple overall heuristics, such that each item can be dealt with effectively.

The mapping upon a logical analogon illuminates the relation among the various operators, as among join, quad, and intersection, between two successive differences and an intersection, or between a select and a calculate. Furthermore, the mapping gives great flexibility in the choice of optimization algorithms and exploits the efficiency of the logical operands and operations.

• *Assumptions*

In the method as described, and as implemented in the prototype, a number of assumptions are made. As stated before, these assumptions are not essential to the basic method since any other set could equally well be applied with appropriate changes in the algorithm. The particular set of assumptions that is used is close to a practical environment in

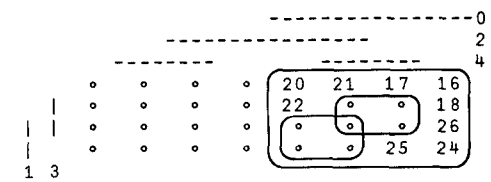
```
 SIZE  COST          COLUMNS
   45        JN      111
  100        CL   2       111
   45   455  JN      111
   90        LD   1       110
   50    10  LD   0 011
 SUM:   465

TB ATT KEY   FR
 0 011 010    50
 1 110 010    90
 2 111 110   100

AT[VAL;]
50 100 100
```



```
F
7

 SIZE  COST          COLUMNS
   45        JN      111
  100        CL   2       111
   45     9  IN      110
   50        RC   2       010
   50        LD   0       011
   90    18  LD   1 110
 SUM:    27
```

Figure 30 Example 7.

```
 SIZE  COST          COLUMNS
    7     6  DF      111
   41   415  IX   2       010
   48        RC   2       010
   48     8  IN           110
   60        LD   4               110
   80    16  LD   3       110
   54    10  DF      111
   22   223  IX   2       010
   22        RC   2       010
   22     5  IN           011
   25        LD   2               011
   43     9  LD   1       011
   95    19  LD   0 111
 SUM:   711

TB ATT KEY   FR
 0 111 100    95
 1 011 010    85
 2 011 010    50
 3 110 100    80
 4 110 100    60

AT[VAL;]
100 50 25
```



```
F
16 17 18 20 21 22 24 25 26

 SIZE  COST          COLUMNS
    7    10  DF      111
   46   467  IX   2       010
   70        UN           010
   48        RC   2               010
   48     6  IN               110
   80        LD   3                       110
   60    12  LD   4               110
   22        RC   2       010
   22     3  IN           011
   43        LD   1               011
   25     5  LD   2       011
   95    19  LD   0 111
 SUM:   522
```

Figure 31 Example 8.

which the method is intended to be applied. We briefly review them here.

The method is kept independent of a potential optimizer that deals with the semantics of the select and calculate expressions. The optimized result, however, clusters the selects that use the same attributes, such that a select optimizer can be used more effectively.

The row by row treatment of the tables is a typical implementation method. It affects the cost calculation used in the optimization. Since this calculation is parameterized, another implementation would require minor adjustments in that calculation. Such a change is localized in the algorithm.

Similarly, a different access method, such as the use of sorting, hashing, or clustered indexes, or a different cost estimate of these methods, results only in local adjustment.

The size estimation depends upon the applicability of the stochastic formulas and the accuracy of the cardinality of the value sets. If a different distribution is known to exist, different formulas and approximations of the size of the expected values will be required. Again, this amounts to a local change, such as the substitution of the pertinent function. If the cardinality is not known, the best available estimate must be used. If the relative importance of these parameters is known, the system may be enhanced by gathering statistical data necessary for more accurate size calculation.

The operator set which is used in the model is relatively extensive to ensure the applicability of the method and seems to indicate that the method is likely to be extendable to other operators of like kind.

The method assumes only first normal form. If higher-order normal form is guaranteed, the size estimation can be improved.

The logical optimization uses the decomposition into prime implicants. Although this is quite effective for the current set of assumptions, it is not an inherent part of the method; any other logical decomposition can be used equally well.

Another assumption is the availability of an index to the key of each table. A deviation from this assumption would result in a change of the cost calculation and possibly in the general structure of the result.

The use of the split is a major feature of the method. Nevertheless, the absence of such a function would still allow the method to be used with profit.

The method uses detailed heuristics at various points. Thus, the duplicate removal is placed at the top of a term; the split is used whenever possible; the partitioning of terms uses an algorithm that favors common terms. All these heuristics can be refined or simplified with a corresponding increase or decrease of computing time for the optimizer. Extensive experience under practical circumstances will quite likely result in various adjustments.

10. Conclusions

The method presented proves to be general and powerful; it is applicable under a great variety of circumstances. The use of a prototype has been invaluable in verifying the correctness of the overall algorithm as well as demonstrating the functioning of the method.

Acknowledgment

This study was performed under the direction of Ir A. J. du Croix, development manager at IBM International Operations Uithoorn, Netherlands. The model was developed at the Technological University Twente, Enschede, Netherlands. The authors wish to thank J. Schoonenberg, director of IBM International Operations Uithoorn, management, and coworkers for the wholehearted cooperation and fruitful discussions. The many useful comments of the referees are gratefully acknowledged.

References

1. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM* **13,** 6, 337–387 (June 1970).
2. D. D. Chamberlin, "Relational Database Management Systems," *ACM Computing Surveys* **8,** 1, 43–66 (March 1976).
3. M. W. Blasgen et al., "System R: An Architectural Overview," *IBM Syst. J.* **20,** 1, 41–62 (January 1981).
4. J. D. Ullmann, *Principles of Database Systems*, Pittman, 1980.
5. J. M. Smith and P. Y. T. Chang, "Optimizing the Performance of a Relational Database Interface," *Commun. ACM* **18,** 10, 568–579 (October 1975).
6. P. A. V. Hall, "Optimization of Single Expressions in a Relational Data Base System," *IBM J. Res. Develop.* **20,** 3, 244–257 (May 1976).
7. A. V. Aho, Y. Saviv, and J. D. Ullman, "Efficient Optimization of a Class of Relational Expressions," *ACM Trans. Database Syst.* **4,** 4, 435–454 (April 1979).
8. J. W. M. Stroet and R. Engmann, "Manipulation of Expressions in a Relational Algebra," *Info. Syst.* **4,** 195–203 (1979).
9. J. Grant and J. Minker, "Optimization in Deductive and Relational Databases," *Advances in Database Theory*, Vol. 1, H. Galaire, J. Minker, and J. M. Nicolas, Eds., Plenum Press, New York, 1981.
10. G. A. Blaauw, A. J. W. Duijvestijn, and R. A. M. Hartmann, "Optimization of Relational Expressions Using a Logical Analogon," Internal Report IBM laboratory, Uithoorn, Netherlands, May 1983.
11. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and P. G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Boston, MA, May 1979, pp. 23–34.
12. P. A. V. Hall and S. J. P. Todd, "Factorisation of Algebraic Equations," *Report UKSC*, IBM United Kingdom Scientific Centre, Peterlee, England, 1974.
13. P. A. V. Hall, "Common Subexpression Identification in General Algebraic Systems," *Scientific Center Report UKSC0060*, IBM Peterlee Scientific Centre, England, 1974.
14. S. Finkelstein, "Common Expression Analysis in Database Applications," ACM SIGMOD International Conference on Management of Data, Orlando, FL, June 1982.
15. M. Karnaugh, "The Map Method of Synthesis of Combinatorial Logic Circuits," *Trans. AIEE* **72,** Part I, 593–598 (November 1953).
16. W. V. Quine, "The Problem of Simplifying Truth Functions," *Amer. Math. Monthly* **59,** 521–531 (October 1952).
17. E. J. McCluskey, "The Minimization of Boolean Functions," *Bell Syst. Tech. J.* **35,** 1417–1444 (November 1956).
18. G. A. Blaauw, A. J. W. Duijvestijn, and A. Ledeboer, "An APL Design Language," Internal Report IBM Laboratory, Uithoorn, Netherlands, May 1979.

Received September 14, 1982; revised April 12, 1983

Gerrit A. Blaauw *Department of Computer Science, Twente University of Technology, Enschede, Netherlands.* Professor Blaauw received the B.S. in electrical engineering from Lafayette College, Easton, Pennsylvania, in 1948 and the Ph.D. in applied science from Harvard University in 1952. While at Harvard, he was on the staff of the Computation Laboratory and participated in the design of the Mark III and Mark IV calculators. From 1952 to 1955, he was on the staff of the Mathematical Center in Amsterdam, Netherlands, where he cooperated in the design of the ARRA and FERTA computers. Dr. Blaauw joined IBM in 1955 at the Poughkeepsie, New York, Product Development laboratory. He was one of the architects of the Stretch computer and of System/360. In 1965, he left IBM to become Professor of Electrical Engineering and Computer Science at the Twente University of Technology in the Netherlands. He is the author of *Digital System Implementation*, written in 1976. In 1979, he received the De Groot Award for his contributions to electrical engineering. Professor Blaauw is a consultant to the IBM World Trade Corporation, a Fellow of the Institute of Electrical and Electronics Engineers, and a member of the Association for Computing Machinery, Sigma Xi, and the Royal Dutch Academy of Science.

A. J. W. Duijvestijn *Department of Computer Science, Twente University of Technology, Enschede, Netherlands.* Professor Duijvestijn is a full professor in the Departments of Computer Science and Electrical Engineering at Twente University of Technology, Enschede, Netherlands, with which he has been associated since 1965. He earned a master's degree in electrical engineering in 1950 from the Technological University, Delft, Netherlands, a master's degree in mathematics in 1955 from the Municipal

University, Amsterdam, Netherlands, and a Ph.D. degree in 1962 from the Technological University, Eindhoven, Netherlands. From 1953 to 1956, Professor Duijvestijn was on the scientific staff of the Computing Department of the Mathematical Center, Amsterdam. From 1956 to 1963, he was on the scientific staff of the Philips Research Laboratory, Eindhoven, Netherlands. He visited IBM in Poughkeepsie, New York, from November 1957 to May 1958 on an exchange program between IBM and Philips. From 1963 to 1965, he was head of the Software Department of Philips Datasystems, Apeldoorn, Netherlands. He visited IBM in Gaithersburg, Maryland, from August 1977 to November 1977. Professor Duijvestijn was President of the Dutch Computer Society from 1977 to 1980. Since 1976, he has been a consultant to IBM International Operations, Uithoorn, Netherlands.

R. A. M. Hartmann *IBM Data Center Services Support Center, Uithoorn, Netherlands.* Mr. Hartmann is an advisory programmer. Before his current assignment, he worked on the IBM 3790 Communication System development and networking. Mr. Hartmann received the M.S. degree in theoretical electrical engineering from the Technological University, Delft, Netherlands, in 1968. He joined IBM in 1970 at the Uithoorn Development laboratory.