International Journal of Computer and Information Sciences, Vol. 10, No. 2, 1981

Evaluation of Queries Based on the Extended Relational Calculi

Isamu Kobayashi[1]

Received June 1980; revised December 1980

An efficient database search algorithm is presented. Four major enhancements on the preceding works have been made. They are (1) relational calculus is extended to enable processing an arbitrary logical function defined on one or more relations, (2) a set of elementary operations which are similar to but are more efficient in processing compound search conditions than the relational algebra is used, (3) the target list processing is completely separated from the search process, and (4) sequential collation procedure is fully utilized to deal with conditions of a certain type defined on two or more relations. The algorithm is composed of two parts: syntactical transformation of the given extended relational calculus and the search execution. Various optimization issues are integrated into these two parts.

KEY WORDS: Database; query evaluation; query optimization; relational algebra; relational calculus; search algorithm.

1. INTRODUCTION

Optimization of the database search operation for the given search condition is an important problem. It grows much more complicated as the search condition becomes complicated. It depends not only on the logical database structure on which the subject search is to be carried out but also on its physical representation in the computer storage and current contents of the database relations.

1.1. Optimization in Three Different phases

Optimization may be achieved in three different phases. One is the selection of optimal database file organization, which is a deliberation

[1] SANNO Institute of Business Administration, School of Management and Informatics. Kamikasuya 1573, Isehara, Kanagawa 259–11, Japan.

extended over a relatively long period of time. Given a specific processing environment, we may determine the file media, physical file organization, and index provisions. The physical database structure thus far selected may not be permanent but is not changed unless a great inexpedience arises.

The second is the selection of optimal search strategy. Usually the given search operation is decomposed into several more elementary operations provided in the system. Optimal decomposition of the given search is achieved by a syntactical analysis of the search conditions with reference to the current database structure. Elementary operations themselves must be carried out as efficiently as possible.

The above search strategy may leave some rooms for the run-time optimization, which selects one specific sequence of elementary operations out of several alternatives with reference to the current contents of database relations and intermediate search results.

In this paper, the last two phases of optimization are discussed. A relational model is used for simplifying description of the algorithm. Notations used in the relational model are slightly modified to enable a simpler description.

1.2. Problem Definition

Database search is the operation which extracts the tuples which are qualified for the given search condition from one or more database relations. The search condition is, in general, of the form

$$x_1 \varepsilon R_1 \wedge x_2 \varepsilon R_2 \wedge \cdots x_N \varepsilon R_N \wedge \lambda(x_1, x_2, \ldots, x_N)$$

with each x_k being a tuple variable belonging to a database relation R_k, and λ being a logical function (whose value is "true" or "false") defined on the Cartesian product of relations, $R_1, R_2, \ldots, R_N$. We will assume that there are no range terms of the form

$$\sim x_k \varepsilon R_k''$$

conjunctively combined with another range term of the form

$$x_k \varepsilon R_k',$$

though they are allowed in Codd's relational calculus. It is because the two can be transformed into a range term

$$x_k \varepsilon R_k$$

after R_k is created by means of a simple set difference operation.

Since the λ is an arbitrary logical function (defined in terms of several attributes of tuple variables), the above search condition is a little more general than Codd's relational calculus. In the relational calculus, λ must be a logical combination (using $\vee$ (or), $\wedge$ (and) and $\sim$ (not) operators) of one or more unit conditions, each being two attribute values or one attribute value and one constant combined by one of relational operators ($=$, $\neq$, $>$, $<$, $\geqslant$ and $\leqslant$). For example,

$$A_1(x) > 3 \times A_2(x)$$

and

$$A_1(x_1) + A_2(x_2) = A_3(x_3),$$

where $A_j(x)$ is the jth attribute of tuple variable x, are disallowed in the relational calculus, while they are allowed as our search conditions. In a pattern matching application, evaluation of search conditions of the form

$$(q, A_i(x))/|q|\,|A_i(x)| > K,$$

where q is a query vector, $A_i(x)$ is an array type attribute, $(q, A_i(x))$ is the inner product of q and $A_i(x)$, $|q|$ and $|A_i(x)|$ are respectively the norm of q and $A_i(x)$ and K is a threshold constant, becomes important. It can also be a search condition though it is not a relational calculus. In this sense, we will call the search condition defined as above an *extended relational calculus*.

Each x_k is a free tuple variable in the extended relational calculus. The extended relational calculus can also contain one or more bound variables. They are bound by universal quantifiers, existential quantifiers or some other means, for example, bound in the scope of ceratain aggregate functions like Σ, average, σ (standard deviation) and maximum. In its prenex normal form, the extended relational calculus can be written as

$$\begin{aligned} & x_1 \varepsilon R_1 \wedge x_2 \varepsilon R_2 \wedge \cdots \wedge x_N \varepsilon R_N \\ & \qquad \wedge \Gamma_1 \Gamma_2 \cdots \Gamma_p(\lambda'(x_1, \ldots, x_N, x_{N+1}, \ldots, x_{N+P}, x_{N+P+1}, \ldots, x_{N+P+Q}), \end{aligned}$$

where x_k is a free tuple variable for $1 \leqslant k \leqslant N$, a quantified tuple variable for $N+1 \leqslant k \leqslant N+P$, and another bound variable for $N+P+1 \leqslant k \leqslant N+P+Q$. Either N, P, or Q can be 0. No free variables exist when $N=0$, no quantified variables exist when $P=0$, and no other bound variables exist when $Q=0$.

Each Γ_k is of the form $\forall x_{N+k} \varepsilon R_{N+k}$ or $\exists x_{N+k} \varepsilon R_{N+k}$, being a shorthand notation of

$$\forall x_{N+k}(\sim x_{N+k} \varepsilon R_{N+k} \vee (\cdots))$$

and

$$\exists x_{N+k}(x_{N+k} \varepsilon R_{N+k} \wedge (\cdots)),$$

respectively, and called a range coupled quantification.

For example, in a search condition

$$x_1 \varepsilon R_1 \wedge \forall x_2 \varepsilon R_2 \left(A_1(x_1) < A_1(x_2) \wedge A_2(x_1) = \sum_{x_3 \varepsilon R_3 \wedge A_3(x_1) = A_3(x_3)} A_4(x_3) \right),$$

$N = 1$, $P = 1$, and $Q = 1$.

We will specify a search operation s with an argument

$$\lambda(x_1, x_2, \ldots, x_N)$$

and names of operand relations $R_1, R_2, \ldots, R_N$, that is,

$$\begin{aligned} &s[\lambda](R_1, R_2, \ldots, R_N) \\ &\quad = \{(x_1, x_2, \ldots, x_N) \mid x_1 \in R_1 \wedge x_2 \varepsilon R_2 \wedge \cdots \wedge x_N \varepsilon R_N \wedge \lambda(x_1, x_2, \ldots, x_N)\}. \end{aligned}$$

Bound variables may appear in the definition of λ but do not leave their vestiges in the search result. Therefore,

$$s[\lambda](R_1, R_2, \ldots, R_N) \subset R_1 \times R_2 \times \cdots \times R_N.$$

The search operation can certainly be achieved by fetching every element in the Cartesian product of $R_1, R_2, \ldots, R_N$ followed by testing it against the given search condition λ. However, this procedure, which is called a *seek* operation, is in most cases too much time-consuming. If the given condition is of some special form (see Sec. 2), we can utilize index provisions prepared in the database and sometimes can utilize efficient sequential collation process, both greatly improving the search efficiency.

In general, the search condition is a logical combination of several unit conditions which cannot be further decomposed into a logical combination of simpler conditions. Since we have

$$s[\lambda_1 \vee \lambda_2](R_1, R_2, \ldots, R_N] = s[\lambda_1](R_1, R_2, \ldots, R_N) \cup s[\lambda_2](R_1, R_2, \ldots, R_N)$$

and

$$s[\lambda_2 \wedge \lambda_2](R_1, R_2, \ldots, R_N) = s[\lambda_1](R_1, R_2, \ldots, R_N) \cap s[\lambda_2](R_1, R_2, \ldots, R_N),$$

the subject search can be achieved by searching with each component condition followed by a union or an intersection operation applied to the intermediate search results. If an efficient search procedure is available for both the searches with λ_1 and λ_2, and the number of qualified elements is considerably small, then the above procedure is faster than the seek on $R_1 \times R_2 \times \cdots \times R_N$.

To be more precise, let p be the number of elements in $R_1 \times R_2 \times \cdots \times R_N$, and q_1 and q_2 be the number of elements among them that are qualified respectively for λ_1 and λ_2. Let s be the time required in fetching an element and t be the time required in testing it against the given condition ($\lambda_1 \wedge \lambda_2$ or $\lambda_2 \vee \lambda_2$) in a seek operation. Finally let u be the time required in fetching an element qualified for the component conditions λ_1 and λ_2 in the available efficient search operation, and v be the time required for each element in making a union or an intersection. (Actually v itself is a function of q_1 and q_2.)

Then the time required for the seek operation becomes $p(s+t)$, while that required for the efficient search operation followed by the union or intersection operation becomes $(q_1+q_2)(u+v)$. In general, $u+v$ is much greater than $s+t$ (t may be negligibly small as compared with s); however, if

$$(s+t)/(u+v) > (q_1+q_2)/p$$

the latter process is faster than the former.

In addition, we have

$$\begin{aligned} s[\lambda_1 \wedge \lambda_2](R_1, R_2, \ldots, R_N) &= s^*[\lambda_1](s[\lambda_2](R_1, R_2, \ldots, R_N)) \\ &= s^*[\lambda_2](s[\lambda_1](R_1, R_2, \ldots, R_N)), \end{aligned}$$

where s^* is the seek operation to be applied to the operand relation which is a subset of a Cartesian product of $R_1, R_2, \ldots, R_N$. If an efficient search process is available for either λ_1 or λ_2, we may shorten the search time by operating the efficient search for one component condition followed by the seek with another condition.

If an efficient search is avaible for λ_1, the time required for the search with λ_1 followed by the seek with λ_2 becomes $q_1(s+t+u)$. Therefore, if

$$(s+t)/(s+t+u) > q_1/p$$

this process improves the whole search efficiency.

Since there may be many logical expressions equivalent to each other, the sequence of operating the search each with a unit condition and the union and intersection operations necessary to achieve the subject search is not uniquely determined. Hence arises a problem of finding a specific operation sequence that brings the best performance into the whole search process.

1.3. Preceding Works

Coping with the problem of finding an optimal decomposition of the search operation into a series of more elementary operations, many improvements on the decomposition process have been reported. If $N = 1$, $s[\lambda](R_1)$ is a general form of "selection" in relational algebra.(4) Astrahan and Chamberlin(1) showed an optimal execution sequence of the selection with a compound search condition. If $N = 2$, $s[\lambda](R_1, R_2)$ is the "join" in relational algebra. Optimization of the join operations was discussed by Smith and Chang(14) and Yao.(20)

Codd(4) had shown that the search operation with his relational calculus can be decomposed into a procedural combination of eight relational algebra operations: union, intersection, difference, projection, selection, join, multiplication, and division. However, no optimization issues were made in relation to his algorithm. Many researchers have presented certain kinds of improvements on Codd's decomposition. Palermo(11) presented an improvement for the search conditions without bound variables. Rothnie(13) and Reiter(10) studied search conditions of a more general form. In INGRES, several improvements were implemented, which were reported by Wong and Youssefi(19) and Held *et al.*(5) In PRTV, sequential collation was introduced for processing some unit conditions.(10)

The following four problems can be pointed out regarding these preceding works.

1. All these works were made for the relational calculus in Codd's sense. It is desirable to extend the algorithm to deal with the extended relational calculus.

2. All these works decomposed the search operation into relational algebra operations, each creating a new relation as an intermediate search result. However, in practical implementations, a great deal of the storage space is used to accommodate intermediate results. There may be another strategy in which only the primary key part (or address) of the qualified tuples or ordered sets of tuples is kept in the intermediate results. After completing the whole search process, the qualified tuple or ordered set of tuples can be fetched either collectively (non-piped mode) or tuple-by-tuple (piped mode) with reference to its primary key value.

3. Since a new relation, that is specified by the target list in the Alpha expression,(3) has to be generated in many cases, it is sometimes attempted to combine a part of generating the new relation by certain relational algebra operations with other relational algebra operations necessary for the given search. However, it is possible only when the target list is of a very simple form. Generating the resultant relation often necessitates a rather

complicated data manipulation, which is better to be left to the care of the employed host language capability.

4. Some preceding works utilized index provisions that were permanently prepared or temporarily created. Some others tried to create sequentially organized files as far as possible to enable efficient sequential processing. However, no preceding works attempted to apply a sequential collation to processing conditions like

$$A_1(x_1) = A_2(x_2) \wedge A_1(x_1) = A_3(x_3)$$

defined for $x_k \varepsilon R_k$ though such a sequential collation is much more efficient than processing component conditions separately followed by joining two resultant relations.

In this paper, we will present a general algorithm, in which the above mentioned problems are resolved. The algorithm is divided into two major phases: syntactical transformation of the search condition and actual search execution. The latter phase is applicable even if the syntactical transformation has not been made.

2. COMPONENT CONDITIONS

From the search process point of view, unit conditions can be classified into two categories: conditions defined on a single relation and conditions defined over two or more relations. The both can be further classified into two subcategories: conditions for which some efficient search process other than the seek operation is available and conditions for which no such efficient process is available. We first review what types of search conditions are classified into what subcategories.

2.1. File Organization

Classification of conditions defined on a single relation is closely related to the file organization employed to represent this relation. Two types of file organizations exist. One is *exclusive*, that means if an exclusive file organization has been employed then any other exclusive file organization cannot be employed. The other is *non-exclusive*, that means a non-exclusive file organization can be employed regardless of what file organization has been employed.

There are a variety of exclusive and non-exclusive file organizations. However, we will count following five exclusive and two non-exclusive file organizations because other organizations are certain modifications of the seven:

(1) Exclusive file organizations.

1-1 Pile file organization that arranges physical records in the sequence of their arrival.

1-2 Sequential file organization that arranges physical records in the sequence of their primary key values.

1-3 Direct file organization that places physical records in the location whose address is calculated from the primary key value.

1-4 Partitioned sequential file organization that divides the sequentially organized file into partitions composed of several consecutive records and provides pointer links among these partitions.

1-5 N-ary tree structured file that integrate n-ary ($n \geqslant 3$) tree search operations into the file organization by providing pointers which direct the record blocks to be fetched next with respect to the primary key value. The B-tree is an example.

(2) Non-exclusive file organizations.

2-1 Binary tree structured file organization is non-exclusive if $n = 2$ because in this case each record block consists of only one record.

2-2 Inverted file (or multilist file) organization that provides an index file separately from the main file.

Except the pile file, all the exclusive file organizations concern a selected *primary key*. On the other hand, non-exclusive file organization can be made with regard to any attribute (including the primary key), which is called a *secondary key*. Details of these file organizations were discussed by Knuth,(9) Martin,(10) Wiederhold,(18) and Kobayashi.(6) Distinct file organizations result in different performance of various search and update operations. However, we will concentrate our discussion into the efficiency of search operations with conditions of some special forms.

2.2. Type *A* and Type *B* Conditions

A search condition defined on a single relation is said to be *type A* if some search procedure being more efficient than the seek operation is available. Otherwise it is called a *type B* condition. Type *A* conditions can be further classified into the following four according to what search procedure is available for improving the search efficiency.

(1) Type $A1$: If a direct file organization is employed with regard to a selected primary key A_p, then conditions of the form

$$A_p(x) = \text{const}$$

can be processed very efficiently. Such a condition is said to be *type A1*. A disjunct of two or more type $A1$ conditions on the same relation can be collectively processed for avoiding duplicate overheads. Hence it may be dealt with as a single type $A1$ condition.

Some conditions could be transformed into a disjunct of type $A1$ conditions. For example, we can transform

$$(A_p(x))^2 - 3 \times A_p(x) + 2 = 0$$

into

$$A_p(x) = 1 \vee A_p(x) = 2$$

However, such a transformation requires a formula manipulation which is not easily processed by a computer program. It may rather be treated as a type $A3$ or type B unit condition deined later.

(2) Type $A2$: If a sequential, index sequential or n-ary tree structured file organization is employed with regard to a selected primary key, then conditions of the form

$$A_p(x)\,\theta\,\text{const}$$

with θ being one of relational operators $=$, $\neq$, $>$, $<$, $\geqslant$, and $\leqslant$ can be processed a little more efficiently than the seek operation. If a binary structured file or an inverted file organization is employed with regard to a secondary ket A_s, then conditions of the form

$$A_s(x)\,\theta\,\text{const}$$

can be processed more efficiently than the seek operation. In these two cases, conditions are said to be *type A2*. A disjunct of two or more type $A2$ conditions defined on the same relation and regarding the same primary or secondary key may be processed collectively to avoid duplicate overhead. Also it is better to process a conjunct of two or more type $A2$ conditions defined on the same relation and regarding the same primary or secondary key collectively. This is particularly effective for conditions like

$$C_1 < A_s(x) \wedge A_s(x) < C_2.$$

Hence a logical (disjunct and/or conjunct) combination of type $A2$ conditions may be dealt with as a single type $A2$ condition.

Some conditions can be transformed into a logical combination of tyep $A2$ conditions. However, since such a transformation requires a formula manipulation which is hard to be implemented as a computer program, it is better to treat them as type $A3$ or type B conditions.

Sometimes a special index file is created for attributes of special type. For example, an inverted index can be made for the set of keywords of document records, which is an array type attribute, and a cluster index can be created for pattern matching.(15,17) The former is used to improve the search with condition

$$\text{keyword } \varepsilon A_s(x)$$

and the latter is used to improve the search with condition

$$\kappa(x) = \text{class}$$

where $\kappa(x)$ is a function of tuple x. These conditions can be regarded as type $A2$ if appropriate index provisions are made.

(3) Type $A3$: If an inverted file organization is employed with regard to a secondary key A_s, then conditions of the form

$$f(A_s(x)),$$

where f is an arbitrary logical function defined in terms of $A_s(x)$ value, can be processed by seeking the index file, which is faster than seeking the main file. Such a condition is said to be *type A3*. During the seek operation on the index file, other type $A2$ and/or type $A3$ conditions regarding the same secondary key can be examined in parallel. Hence a logical combination of a type $A3$ condition and other type $A2$ and/or type $A3$ conditions regarding the same secondary key may be dealt with as a single type $A3$ condition.

(4) Type $A4$: If two inverted file organizations, one regarding a secondary key A_s and the other regarding another secondary key A'_s, are employed at the same time, condition

$$A_s(x) = A'_s(x)$$

can be processed by a sequential collation of two index files (index files are usualy organized enabling sequential accessing with regard to the secondary key). If the matched index entries contain a common value pointing a tuple in the main file, it directs a qualified tuple. The condition is said to be *type A4*. During the sequential collation of the two index files, other type $A2$

and/or type $A3$ conditions defined on the same relation and regarding either A_s or A'_s can be examined in parallel. Hence a logical combination of a type $A4$ condition and other type $A2$ and/or type $A3$ conditions regarding either one of the secondary keys may be dealt with as a type $A4$ condition. A logical combination of two type $A4$ conditions can be processed in parallel if the two share a common secondary key. For example,

$$A_s(x) = A'_s(x) \wedge A_s(x) = A''_s(x)$$

can be processed in parallel by sequentially collating three index files provided for A_s, A'_s, and A''_s. Therefore, it may be dealt with as a single type $A4$ condition.

(5) Type B: All the conditions that are not type A ($A1, A2, A3$, or $A4$) are said to be *type* B. A type B condition must be processed by a seek operation. A disjunct of a type B condition and type A and/or type B conditions all defined on the same relation can be processed in parallel during the seek operation. Hence it may be dealt with as a type B condition. A conjunct of type B conditions defined on the same relation may also be dealt with as a single type B condition.

Shown below are examples of type A and type B conditions thus far defined.

1. $A_p(x) = 123$, $A_p(x) = \text{Kobayashi} \vee A_p(x) = \text{Yokomori}$—type $A1$ if A_p is the primary key used for a direct file.

2. $A_p(x) = 123$, $A_p(x) > 123 \wedge A_p(x) < 234$—type $A2$ if A_p is the primary key used for a (or a partitioned) sequential or an n-ary tree structured file.

3. $A_s(x) < 123$, $A_s(x) = \text{manager} \vee A_s(x) = \text{supervisor}$—type $A2$ if A_s is a secondary key.

4. $A_s(x)^2 + 3 \times A_s(x) > 10$, $\sin(A_s(x) + \pi) < 2 \times \cos(A_s(x))$—type $A3$ if A_s is a secondary key and indexed by index files.

5. $A_s(x) = A'_s(x)$, $A_s(x) = A'_s(x) \wedge A_s(x) = A''_s(x) \wedge A''_s(x) < 123$—type $A4$ if all A_s, A'_s, and A''_s are secondary key indexed by index files.

6. $A_q(x) = 123$—type B if A_q is neither a primary key nor a secondary key.

7. $A_r(x) < A'_r(x)$, $A_r(x) + A'_r(x) = A''_r(x)$, $A_r(x) \times A'_r(x) = 24 \vee A''_r(x) = 123$—type B regardless of whether A_r, A'_r, and A''_r are (primary or secondary) keys or not.

Let u_k be the time required for fetching a qualified tuple using the available efficient search procedure for type Ak conditions (it varies according to the cases but we can estimate an average time) and s, the time required for fetching a tuple and testing against the search condition in the seek procedure. Then it seems safe to say

$$s < u_1 < u_2 < u_3 < u_4.$$

Let p be the number of tuples in the relation to which the subject search is to be applied and q be the number of quantified tuples in them. Then the available efficient search procedure for type Ak conditions is more efficient than the seek operation if

$$s/u_k > q/p.$$

In general, q is not known before the search. However, in most cases, if p is sufficiently large, q/p can be estimated to be smaller than s/u_k. Conversely, if p is small, q/p can be greater than s/u_k, and hence the seek operation is better. The s/u_k value depends on the specific implementation of the search procedure.

The above classification is by no means an absolute one. Certain advanced file organizations can be devised in future. Also some hardware devices (database machines) can be invented in future that brings a better performance into processing some types of search conditions. In such cases, the classification we have mentioned can be modified accordingly.

2.3. Sequential Collation

We have already seen that a sequential collation of two index files can be applied to processing the condition

$$A_s(x) = A'_s(x)$$

defined on a relation. The same technique can be applied to processing the condition

$$A_s(x_1) = A'_s(x_2)$$

defined for $(x_1, x_2) \varepsilon R_1 \times R_2$.

Let p_1 and p_2 be the number of tuples respectively in R_1 and R_2, and s the time required for fetching a tuple and testing it against the given condition in sequential processing. The total time required for seeking $R_1 \times R_2$ is about $p_1 p_2 s$. On the other hand, the time required for sequential collation of the two files is $(p_1 + p_2)s$. Except when either one of R_1 and R_2 is empty or includes only one tuple, we have

$$p_1 + p_2 < p_1 p_2.$$

The latter process may necessitate a sorting procedure before executing the collation; however, since the time required for sorting p tuples is in the order of $p \log p$, the latter process is better than the former when p_1 and p_2 are sufficiently large.

In addition, the sequential collation can be operated on two index files one for A_s and the other for A'_s, instead of being operated on two main files. If two such index files have already been provided, they can be used as they are. Even if either or both index files have not been provided, we can temporarily generate them. This may greatly reduce the time required for sorting and collating tuples.

2.4. Type *C* and Type *D* Conditions

Search conditions defined over two or more relations can be classified into the type C or type D conditions defined as follows according to whether a sequential collation can be applied or not.

(6) Type C: Conditions of the form

$$A_s(x_1) = A'_s(x_2)$$

defined for $(x_1, x_2) \varepsilon R_1 \times R_2$ is said to be *type* C. A conjunct of m type C conditions of the form

$$\bigwedge_{k=2}^{m} (A_s^{(1)}(x_1) = A_s^{(k)}(x_k))$$

or a conjunct equivalent to the above can be processed in a single sequential collation. Hence it can be dealt with as a type C condition. Some attributes involved in a type C condition may have already been indexed by an index file, while others have not been indexed and temporary index file must be created for each of them before the sequential collation. If type $A2$, $A3$, and/or $A4$ conditions regarding one of the indexed attributes are conjunctively combined with the subject type C condition, they can be processed in parallel during the sequential collation process. Hence they, together with the type C condition, can be dealt with as a type C condition. On the other hand, if type B conditions defined on one of the involved relations, for which a temporary index file must be created before the sequential collation, are conjunctively combined with the subject type C condition, they can be processed in parallel during the generation of the temporary index files. Hence they, together with the type C condition, may be dealt with as a type C condition.

(7) Type D: All other unit conditions defined over two or more

relations are said to be *type* D. A type D condition must be processed by seeking the Cartesian product of the involved relations. A disjunct of unit conditions defined on the same set of relations, at least one of which is a type D condition, can be processed during a single seek operation. Hence it may be dealt with as a type D condition. A conjunct of type D conditions all defined on the same set of relations can be processed during a single seek operation. Hence it may be dealt with as a type D condition.

Shown below are examples of type C and type D conditions thus far defined.

1. $A_s(x_1) = A'_s(x_2)$, $A_s(x_1) = A'_s(x_2) \wedge A_s(x_1) = A''_s(x_3) \wedge A''_s(x_3) < 123$—type C if all A_s, A'_s, and A''_s are indexed by index files.

2. $A_r(x_1) - A'_q(x_2)$, $A_r(x_1) = A_q(x_2) \wedge A_q(x_2) = A'_r(x_3) \wedge A_q(x_2) < 123$—type C if A_q is not indexed regardless of whether A_r and A'_r are indexed or not.

3. $A_r(x_1) < A'_r(x_2)$, $A_r(x_1) + A'_r(x_2) = A''_r(x_3)$, $A_r(x_1) = A''_r(x_2) \vee A'_t(x_1) < A'''_r(x_2)$—type D regardless of whether A_r, A'_r, A''_r, and A'''_r are indexed or not.

The sequential collation for type C conditions is much more efficient than the seek operation applied to the Cartesian product of relations except when $m - 1$ of m involved relations contain only a very small number of tuples. Care should be taken not to apply sequential collation to m relations one of which is empty.

2.5. Unit Conditions with Aggregate Functions

Unit conditions containing aggregate functions must be dealt with as type B or type D conditions according to their being defined on a single relation or on more than one relation. During the seek operation, these aggregate functions must be evaluated by invoking an appropriate procedure. Such a procedure includes the search with the condition which determines the range of aggregation. Hence in this case a recursive execution of the search procedure must be made.

We will not go into the details of search procedures processing various types of search conditions because they vary greatly according to the employed hardware and file organizations. Diversity of implementations results in various performances. Instead we will discuss selection of an optimal sequence of elementary search and other operations for processing the given compound search conditions.

3. SYNTACTICAL TRANSFORMATION

The first phase of search optimization is the syntactical transformation of the given compound search condition. It can be done at the compiling time probably with an aid of an appropriate syntax parser equipped in the employed programming language translator (host language compiler). The purpose of this syntactical transformation is to move the unit or compound component conditions, for which some efficient search procedure is available, to a more forehand position in each conjunctive term. This narrows down the area on which a seek operation must be made and, in consequence, improve the total search efficiency.

To enable such a syntactical transformation, it is advisable that the given condition is transformed into its prenex form, and then its matrix part is transformed into a conjunctive normal form. It is better to apply some remaining procedures to a disjunctive normal form.

The syntactical transformation can be achieved by executing the following ten steps for the λ argument of the search operation.

STEP 1: Transformation into the prenex normal form. We transform the given condition into its prenex normal form. If it contains no free tuple variable, a free tuple variable whose domain is an arbitrary relation that includes only one arbitrary tuple is added. A logical function defined on the added relation whose value is constantly "true" is conjunctively combined with the given condition.

The first half of this step assembles all the quantifications appearing in the given condition in front of the matrix part. The last half is necessary to deal with conditions without free variables. Such a condition becomes to have to be evaluated, for instance, in integrity checking at the database update.

For example, step 1 transforms.

$$\forall x \varepsilon R_2(\lambda_1(x)) \vee \forall x \varepsilon R_2(\lambda_2(x))$$

into

$$\forall x_2 \varepsilon R_2 \, \forall x_3 \varepsilon R_3(T(x_1) \wedge (\lambda_1(x_2) \wedge (\lambda_2(x_3)))$$

where x_1 is an artificially added tuple variable whose range in an arbitrary relation R_1 containing only one tuple, and T is a constant function whose value is "true."

As the result, we obtain the form

$$\Gamma_1 \Gamma_2 \cdots \Gamma_P(\lambda(x_1, \ldots, x_N, x_{N+1}, \ldots, x_{N+P}, \ldots, x_{N+P+Q})),$$

where x_k is a free variable for $1 \leqslant k \leqslant N$ $(N \geqslant 1)$, a quantified variable for $N+1 \leqslant k \leqslant N+P$ $(P \geqslant 0)$ and another bound variable for $N+P+1 \leqslant k \leqslant N+P+Q$ $(Q \geqslant 0)$.

STEP 2: Transformation into the conjunctive normal form. We transform the matrix part of the condition obtained in step 1 further into its conjunctive normal form.

This step is not absolutely necessary but is desirable in order to examine if there exists some disjunct of unit conditions to be regarded as a single condition.

STEP 3: Elimination of negation operators. For all the component conditions of the form

$$\sim(f\theta g)$$

appearing in the matrix part, we remove the "$\sim$" operator by changing the relational operator θ to another relational operator θ' appropriately.

For example, "$=$" is replaced by "$\neq$," "$>$" is replaced by "$\leqslant$," and "$\leqslant$" is replaced by "$>$." If a component condition prefixed by a "$\sim$" operator is not of the above form (the case when a component condition is a logical function defined is some way other than by a relational operator), it should remain unchanged but is hereafter dealt with collectively as a unit condition during the syntactical transformation.

STEP 4: Determination of unit condition type. We determine the type of every unit condition in the matrix part of the transformed condition.

Unit conditions are classified into type $A1$, $A2$, $A3$, $A4$, B, C, and D conditions. If conventional notations like

$$C_1 < A_1(x) < C_2,$$

$$A_1(x) = A_2(x) = A_3(x)$$

and

$$A_1(x_1) = A_2(x_2) = A_3(x_3) = A_4(x_4)$$

are allowed, they might be decomposed into a conjunct of several unit conditions. However, they may rather be regarded as a unit condition of a certain type.

STEP 5: Redefinition of the type of disjuncts. We examine each disjunctive term of the conjunctive form to find the disjunct to be dealt with as a single condition of a certain type according to the following rules.

(1) If there are more than one type $A1$ condition which are defined on the same relation, they are collectively refefined as a type $A1$ condition.

(2) If there are more than one type $A2$ condition which are defined in terms of the same primary or secondary key on the same relation, they are collectively redefined as a type $A2$ condition.

(3) If there are a type $A3$ condition and several type $A2$ and/or $A3$ conditions all defined in terms of the same secondary key on the same relation, they are collectively redefined as a type $A3$ condition.

(4) If there are a type $A4$ condition and several type $A2$, $A3$, and/or $A4$ conditions all defined in terms of secondary keys, at least one of which is the same key, on the same relation, they are collectively redefined as a type $A4$ condition.

(5) If there are several condition defined on the same relation, at least one of which is type B, they are collectively redefiend as a type B condition.

(6) If there are several conditions defined over the same set of relations, at least one of which is type D, they are collectively redefined as a type D condition.

This step is repeated until the disjuncts to which the above rules are applicable have been exhausted.

The six rules reflect the type issues we made in Sec. 2.

This step can be achieved easily by arranging unit conditions in each disjunctive term according to the names of the involved relations and, if given, the names of the involved attributes. Since rule (k) in this step overrides rule (k-1), we had better test rule (6) first and (1) last. Rules (2), (3), and (4) can be processed collectively for type $A2$, $A3$, and $A4$ conditions defined on the same relation. This procedures resembles the procedure of obtaining connected components of a graph.

STEP 6: Transformation into the disjunctive normal form. We transform the condition obtained in step 5 into its disjunctive normal form.

Again we need a relatively simple formula manipulation procedure. The number of component conditions may have been decreased as the result of step 5 as compared with the number of unit conditions to be processed in step 2.

STEP 7: Factoring out common conditions. If there exists a common component condition being given a type in two or more conjunctive terms that are disjunctively combined without any interlaid parentheses, we parenthesize these terms and factor out the common condition in front of the parentheses.

This step is repeated until such common component conditions have been exhausted.

Since there may be more than one such common condition in conjunctive terms, the result of the above procedure is not unique. We will introduce a heuristics in the sequence of picking up such common conditions. The following sequence seems a suitable selection for narrowing down the search area as rapidly as possible:

a) Type $A1$ conditions.

b) Type $A2$ conditions. If the number of qualified tuples for the condition is known (for instance, by means of appropriate information obtainable from the index file), the ascending sequence of this number is appropriate for picking up type $A2$ conditions.

c) Type $A3$ conditions.

d) Type $A4$ conditions.

e) Type C conditions. The ascending sequence of the number of involved relations is appropriate for picking up type C conditions.

f) Type B, conditions.

g) Type D conditions. The ascending sequence of the number of involved relations is appropriate for picking up type D conditions.

As the result, we may have a nest of several parentheses. For example, if we are given a condition

$$\lambda_1 \wedge \lambda_2 \wedge \lambda_3 \vee \lambda_1 \wedge \lambda_2 \wedge \lambda_4 \vee \lambda_1 \wedge \lambda_5 \wedge \lambda_6,$$

it is transformed into

$$\lambda_1 \wedge (\lambda_2 \wedge (\lambda_3 \vee \lambda_4) \vee \lambda_5 \wedge \lambda_6).$$

During the factoring out procedure, it may happen that all component conditions in a conjunctive term are factored out. Then this term together with other terms disjunctively combine with it and "V" operators combining them are simply deleted. For example,

$$\lambda_1 \wedge \lambda_2 \wedge \lambda_3 \vee \lambda_1 \wedge \lambda_3$$

is simply transformed into

$$\lambda_1 \wedge \lambda_3.$$

The assotiative law regarding "$\wedge$" operators is used for avoiding unnecessary generation of parentheses. For example,

$$\lambda_1 \wedge (\lambda_2 \wedge (\lambda_3 \vee \lambda_4))$$

is written as

$$\lambda_1 \wedge \lambda_2 \wedge (\lambda_3 \vee \lambda_4).$$

Step 7 enables to eliminate duplicate evaluations of the same condition that might be executed unless such a factoring out procedure had been provided.

STEP 8: Redefinition of the type of conjuncts. We examine each conjunctive term of the disjunctive form to find the conjunction to be dealt with as a single condition of a certain type according to the following rules.

(1) If there are more than one type $A2$ condition which are defined in terms of the same primary or secondary key on the relation, they are collectively redefined as a type $A2$ condition.

(2) If there are a type $A3$ condition and several type $A2$ and/or $A3$ conditions all defined in terms of the same secondary key on the same relation, they are collectively redefined as a type $A3$ condition.

(3) If there are a type $A4$ condition and several type $A2$, $A3$, and/or $A4$ conditions all defined in terms of secondary keys, at least one of which is the same key, on the same relation, they are collectively redefined as a type $A4$ condition.

(4) If there are several type B conditions defined on the same relation, they are collectively redefined as a type B condition.

(5) If there are two type C conditions, in which one attribute appears in common, are collectively redefined as a type C condition. This redefinition must be applied repeatedly.

(6) If there are a type C condition and several type $A2$, $A3$, and/or $A4$ conditions defined in terms of the secondary keys, one of which is the secondary key appearing in this type C condition, they are collectively redefined as a type C condition.

(7) If there are a type C condition and several type B conditions defined in terms of an attribute, which appear in this type C condition, they are collectively redefined as a type C condition.

(8) If there are several type D conditions defined over the same set of relations, they are collectively redefined as a type D condition.

The eight rules also reflect the type issues we made in Sec. 2.

This step can be achieved also by arranging unit conditions in each conjunctive terms according to the names of the involved relations and, if given, the names of the involved attributes. Different from step 5, we had better apply these rules in the ascending sequence of their numbering. Rules (1) through (3) can be processed collectively for type $A2$, $A3$, and $A4$ conditions. Also rules (5) through (7) can be processed collectively when one or more type C conditions exist.

STEP 9: Determining the type of compound conditions enclosed in parentheses. We add a pair of parentheses enclosing the whole matrix part, and then examine each compound condition enclosed in a pair of parentheses form the innermost to the outermost parentheses. Compound condition type is determined according to the following rules.

(1) If all component conditions in a pair of parentheses are type A ($A1$, $A2$, $A3$, $A4$) or type A^*, then the compound condition is type A^*.

(2) If all component conditions in a pair of parentheses are type A, A^*, C, or C^*, then the compound condition is type C^*.

(3) If all component conditions in a pair of parentheses are type A, A^*, C, C^*, B, or B^*, then the compound condition is type B.

(4) If at least one component condition in a pair of parentheses is type D or type D^*, then the compound condition is type D^*.

Note that the type is determined without regard to what logical operators are used for combining component conditions.

The above step includes a recursive procedure.

STEP 10: Rearrangement of component conditions. We finally rearrange all component conditions directly combined by "$\wedge$" operators in the sequence of

$$A1\text{–}A2\text{–}A3\text{–}A4\text{–}A^*\text{–}C\text{–}C^*\text{–}B\text{–}B^*\text{–}D\text{–}D^*.$$

The rearrangement must be applied to compound conditions in each pair of parentheses.

Again we have employed a heuristics in the above arrangement to enable narrowing down the search area as rapidly as possible.

Figure 1 shows an example of syntactical transformation applied to a compound condition defined on a relation. Only the transformation of the matrix part is shown. The illustrated condition graph may ease understanding. Figure 2 shows an example in which a compound condition defined over three relations is transformed.

Assume $\lambda_1, \lambda_3, \lambda_4, \lambda_5, \lambda_6$: type A ($\lambda_5$ and λ_6 being type A2 defined in termes of the same attribute), λ_2: type B.

Given condition

$(\lambda_1 \vee \lambda_2) \wedge (\lambda_3 \vee \lambda_4 \vee (\lambda_5 \wedge \lambda_6))$

Conjunctive normal form

$\lambda_1 \vee \lambda_2 \wedge \lambda_3 \vee \lambda_4 \vee \lambda_5 \wedge \lambda_3 \vee \lambda_4 \vee \lambda_6$

Redefinition of condition type

$\lambda_1 \vee \lambda_2 \equiv \mu_1$: $\mu_1 \wedge \lambda_3 \vee \lambda_4 \vee \lambda_5 \wedge \lambda_3 \vee \lambda_4 \vee \lambda_6$

Disjunctive normal form

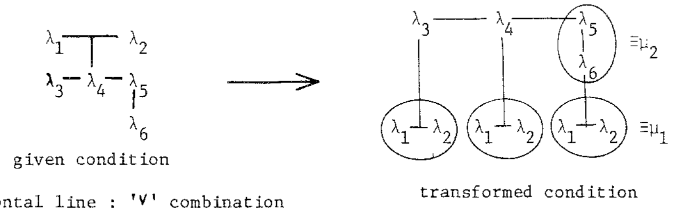
$\mu_1 \wedge \lambda_3 \vee \mu_1 \wedge \lambda_3 \wedge \lambda_4 \vee \mu_1 \wedge \lambda_3 \wedge \lambda_5 \vee \mu_1 \wedge \lambda_3 \wedge \lambda_6 \vee \mu_1 \wedge \lambda_4 \vee \mu_1 \wedge \lambda_4 \wedge \lambda_5 \vee \mu_1 \wedge \lambda_4 \wedge \lambda_6 \vee \mu_1 \wedge \lambda_5 \wedge \lambda_6$

Factoring

$\lambda_3 \wedge \mu_1 \vee \lambda_4 \wedge \mu_1 \vee \lambda_5 \wedge \lambda_6 \wedge \mu_1$

Redefinition of condition type

$\lambda_5 \wedge \lambda_6 \equiv \mu_2$: $\lambda_3 \wedge \mu_1 \vee \lambda_4 \wedge \mu_1 \vee \mu_2 \wedge \mu_1$



horizontal line : 'V' combination

vertical line: '$\wedge$' combination

Fig. 1. Syntactical transformation applied to a condition defined on one relation.

Assume λ_{A1}: type A defined on R_1, λ_{A2}: type A defined on R_2, λ_{B3}: type B defined on R_3, λ_{C12}: type C defined on $R_1 \times R_2$, λ_{C23}: type C defined on $R_2 \times R_3$ (λ_{C12} and λ_{C23} regard the same attribute of R_2), λ_{D13} type d defined on $R_1 \times R_3$.

Given condition

$\lambda_{A2} \wedge \lambda_{D13} \wedge \lambda_{C12} \wedge \lambda_{A1} \wedge \lambda_{C23} \wedge \lambda_{B3}$

Redefinition of condition type

$\lambda_{C12} \wedge \lambda_{C23} \wedge \lambda_{B3} \equiv \mu_{C123}$

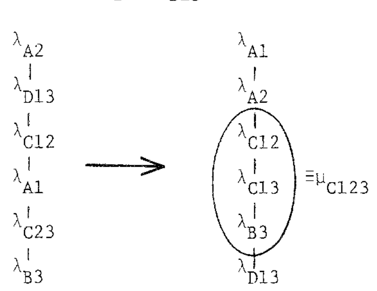


Fig. 2. Syntactical transformation for a condition defined on $R_1 \times R_2 \times R_3$.

4. ELEMENTARY OPERATIONS

In order to process conditions and other symbols in the transformed condition, we need several elementary operations. They resemble operations in the relational algebra[1] but differ from relational algebra operations in that relational algebra operations always generate a new relation, while our elementary operations generate a subset of operand relation or a subset of the Cartesian product of operand relations. Theoretically a subset of the Cartesian product of relations can be regarded as a relation but it is better to distinguish the two from the software implementation point of view. In fact, the elementary operations need not generate a new relation as the result. Only the (ordered) sets of primary keys of the tuples (or addresses of the records representing them) that are components of elements in the result must be kept.

The following six must be provided as elementary operations.

(1) *Extended join*: $j[\lambda](R_1, R_2, \ldots, R_m)$. The extended join is the $s[\lambda](R_1, R_2, \ldots, R_m)$ operation with λ being a unit condition or a compound condition, which is given a type ($A1$, $A2$, $A3$, $A4$, B, C or D) during the syntactical transformation procedure. Its optimal implementation varies according to the condition type. We will assume that an optimal implementation has been made for all condition types as discussed in Sec. 2, with the given hardware and database organization taken into account.

If $m = 1$, the extended join is identical to the selection in relational algebra. If $m = 2$, it resembles the join in relational algebra; however, the result is not a relation but a subset of $R_1 \times R_2$. Generally, the result is a subset of $R_1 \times R_2 \times \cdots \times R_m$.

(2) *Extended selection*: $r[\lambda](S)$. The extended selection is defined by

$$r[\lambda](S) = \{(t_1, t_2, \ldots, t_m) | (t_1, t_2, \ldots, t_m) \varepsilon S \wedge \lambda(t_1, t_2, \ldots, t_m)\},$$

where S is a subset of $R_1 \times R_2 \times \cdots \times R_m$, which is obtained as an intermediate search result. This operation must be executed as a seek operation on S.

(3) *Extended intersection*: $i(S_1, S_2)$. The extended intersection is defined by

$$i(S_1, S_2) = \{(t_1, t_2, \ldots, t_m) \mid (t_{i1}, t_{i2}, \ldots, t_{im'}) \varepsilon S_1 \wedge (t_{j1}, t_{j2}, \ldots, t_{jm''}) \varepsilon S_2\},$$

where

$$\{i1, i2, \ldots, im'\} \cup \{j1, j2, \ldots, jm''\} = \{1, 2, \ldots, m\},$$

and S_1 and S_2 are, respectively, a subset of $R_{i1} \times R_{i2} \times \cdots \times R_{im'}$ and $R_{j1} \times R_{j2} \times \cdots \times R_{jm''}$. The extended intersection operations may be implemented in several different ways according to the cases.

If

$$\{i1, i2,\ldots, im'\} \cap \{j1, j2,\ldots, jm''\} = \{k1, k2,\ldots, km'''\}$$

is not an empty set, the extended intersection resembles the natural join in relational algebra. In particular, if

$$\{i1, i2,\ldots, im'\} = \{j1, j2,\ldots, jm''\} = \{1, 2,\ldots, m\},$$

the extended intersection is the intersection in set theory. In these cases, the extended intersection can be carried out by a sequential collation of two sets of ordered set of tuples, each being sorted by the concatenation of the primary key values of tuples in $(t_{k1}, t_{k2},\ldots, t_{km'''})$.

If

$$\{i1, i2,\ldots, im'\} \cap \{j1, j2,\ldots, jm''\} = \phi$$

the extended intersection implies making the Cartesian product of S_1 and S_2. Although it becomes a time and space-consuming task, implementation of the operation for this case will be easy. (The user may be given a warning message when such a time and space consuming procedure is invoked.)

(4) *Extended union*: $u(S_1, S_2)$. The extended union is defined by

$$u(S_1, S_2) = \{(t_1, t_2,\ldots, t_m) \mid (t_1, t_2,\ldots, t_m) \varepsilon S_1 \vee (t_1, t_2,\ldots, t_m) \varepsilon S_2\},$$

where both S_1 and S_2 are a subset of $R_1 \times R_2 \times \cdots \times R_m$. To eliminate duplicate elements, a sorting procedure with the concatenation of primary key values of tuples in each element of S_1 and S_2 following by a merging procedure is mandatory.

(5) *Extended projection*: $p[i1, i2,\ldots, im'](S)$. The extended projection is defined by

$$p[i1, i2,\ldots, im'](S) = \{(t_{i1}, t_{i2},\ldots, t_{im'}) \mid (t_1, t_2,\ldots, t_m) \varepsilon S\},$$

where

$$\{i1, i2,\ldots,im'\} \subset \{1, 2,\ldots, m\}$$

and S is a subset of $R_1 \times R_2 \times \cdots \times R_m$. Again a sorting procedure must be integrated for removing duplicate elements in the result.

(6) *Extended division*: $v(S, R_m)$. The extended division is defined by

$$v(S, R_m) = \left\{ (t_1, t_2, \ldots, t_{m-1}) \mid \forall t'_m \varepsilon R_m \, \exists (t''_1, t''_2, \ldots, t''_m) \varepsilon S \left(\bigwedge_{j=1}^{m-1} (t_j = t''_j) \wedge t'_m = t''_m \right) \right\},$$

where S is a subset of $R_1 \times R_2 \times \cdots \times R_m$. This operation can be carried out by (i) sorting all elements $(t''_1, t''_2, \ldots, t''_m)$ is S by the concatenation of primary key values of component tuples, (ii) sorting tuples in R_m by their primary key values, and (iii) for every subset S' of S in which all elements have a common $(t_1, t_2, \ldots, t_{m-1})$, collating S' and R_m sequentially to examine if $p[m](S')$ covers R_m. If so, corresponding $(t_1, t_2, \ldots, t_{m-1})$ is placed into the result. If not, it is excluded from the result.

Table I. Rough Estimation of the Time Required for Elementary Operations

Join		
	type $A1$	aq
	type $A2$	$b' \log p \sim bp$
	type $A3$	bp
	type $A4$	mbp
	type B	cp
	type C	$b \sum_i p_i \sim d \sum_i p_i \log p_i + b \sum_i p_i$
	type D	$c \prod_i p_i$
	$p(p_i)$: number of tuples in the operand relation	
	q: number of qualified tuples in the operand relation	
	m: number of involved attributes	
	a, b, b', c, d: implementation dependent constant ($c = c'l$ and $d = d'l$ with l being the length of records representing tuples).	
Selection		ner
	n: number of the involved relations	
	r: number of elements in the operand	
	e: implementation dependent constant	
Intersection		
	natural join/intersection	$f(r_1 \log r_1 + r_2 \log r_2) + g(r_1 + r_2)$
	Cartesian product	$gr_1 r_2$
	r_1, r_2: number of elements in each operand	
	f, g: implementation dependent constant	
Union		$f(r_1 \log r_1 + r_2 \log r_2) + g(r_1 + r_2)$
Projection		$fr \log r$
Division		$f(r \log r + p \log p) + 2gr$
	r: number of elements in the first operand	
	p: number of elements in the second operand	

As mentioned previously, the results of these elementary operations do not necessarily contain all the attribute values of qualified tuples or qualified ordered sets of tuples. In fact, except the extended join and extended selection, all our elementary operators can be carried out using only the primary key part of tuples in the intermediate results. Even when the other attribute values become necessary in executing an extended selection, they can be availed using the primary key part of tuples. Therefore, we have to keep only the primary key (or address) values of the resultant tuples for the further processes. In this way, the data volume to be moved among the storage devices as well as the storage space to accommodate intermediate results is minimized. This also saves the time. From the software implementation point of view, this is the basic difference between relational algebra operations and our elementary operations.

Rough estimation of necessary time for each elementary operation is shown in Table I. Although it includes many factors that are implementation-dependent, we can see what the order of time requirement is in terms of the number of elements in the operand.

The above operations can be implemented software-wise but they may also be embodied as some special database machines. We have seen that in all the extended join for type C condition, extended intersection except when it implies making a Cartesian product, extended union, extended projection, and extended division, the sorting serves as a basic procedure. A sorting machine, if it were implemented hardware-wise, might become a very powerful tool for efficient implementation of these operations.

5. THE RECURSIVE SEARCH PROCEDURE

We are now ready to execute actual search operations. Since the syntactically transformed condition may contain a nest of parentheses, the search procedure must be written in a recursive manner. Each invocation of the recursive procedure deals with a compound condition enclosed in a pair of parentheses.

In order to process the given condition that is defined on a Cartesian product of N relations, we provide a set of $2^N - 1$ workspaces each being able to accommodate a name of intermediate result (by which all the elements in the intermediate result can be accessed) and the number of elements in it. These workspaces are so numbered that content of a workspace specifies a subset of the Cartesian product of relations which are specified by its number (Table II).

For example, if $N = 1$, we have to provide only one workspace ws[1]. If $N = 2$, we provide $2^2 - 1 = 3$ workspaces, ws[1], ws[2] and ws[1, 2]. If $N = 3$, we provide $2^3 - 1 = 7$ workspaces, ws[1], ws[2], ws[3], ws[1, 2],

Table II. Workspace Identification

Workspace	Name and number of elements in it
ws[i]	a subset of R_i
ws[$i1, i2$]	a subset of $R_{i1} \times R_{i2}$
ws[$i1, i2, i3$]	a subset of $R_{i1} \times R_{i2} \times R_{i3}$
....	....
ws[$1, 2, \dots, N$]	a subset of $R_1 \times R_2 \times \cdots \times R_N$

ws[1, 3], ws[2, 3], and ws[1, 2, 3]. Actually a binary numbering in which each bit corresponds to a relation can be adopted.

Each invocation of the recursive search procedure uses a subset of these $2^N - 1$ workspaces. If the search procedure is to deal with a compound condition defined on a Cartesian product of n ($n \leqslant N$) relations, it uses $2^n - 1$ workspaces. In order to process a disjunct of conditions, several intermediate states of these workspaces must be kept. Hence we provide a push-down stack, which we call the *X stack*, each of whose entries accommodate a copy of the $2^n - 1$ workspaces. Also we must provide a temporary area called the *Y area*, that accommodates a copy of the workspaces. Finally one more push-down stack, which we call the *Z stack*, must be provided. Each entry of the Z stack accommodates a name of intermediate result and the number of elements in it.

The compound condition in a pair of parentheses is examined from left to right and component conditions are processed one by one. Each time a component condition has been processed, contents of the workspaces are updated accordingly. When a left parenthesis has been encountered, the search procedure is invoked recursively.

The formal parameter of the search procedure must include

n: *integer*; *function* λ: *Boolean*; *var* ws: *array*[1..2***n* − 1] *of* workspace;

where λ is the compound condition to be processed. Several local variables including the X stacks, Y area, and Z stack must be defined in the search procedure.

We will describe the search procedure in a PASCAL-like pseudoprogramming language. The following program shows the skeleton of the procedure.

```
procedure search
  begin Push down the workspace into the X stack;
        Store the workspace into the Y area;
    Δ While there remain some units to be processed do
    begin Get next unit;
            If the obtained unit is a component condition then
      begin Pop up the $X$ stack into the workspace;
            Invoke procedure "preprocess";
            Invoke procedure "conditionprocess";
            Invoke procedure "postprocess";
            Push down the workspaces into the $X$ stack
      end
            Else if the obtained unit is a left parenthesis then
      begin Find the corresponding right parenthesis;
            Extract the compound condition enclosed in this pair of
            parentheses;
            Invoke procedure "search" recursively
      end
            Else if the obtained unit is an "$\vee$" operator then
              Push down the content of $Y$ area into the $X$ stack
    end;
        Invoke procedure "union";
        Release all the storage areas used for accommodating inter-
        mediate results in this procedure
  end search;
```

The Δ is the label for establishing a return point from procedure "eliminate" that is described later.

We will next describe several procedures that are invoked in the search procedure.

5.1. The Preprocess Procedure

The preprocess procedure is necessary to find the operand to which the extended join or extended selection is to be applied for processing the given component condition. Let the given condition be μ or $\sim\mu$ defined on $R_{i1} \times R_{i2} \times \cdots \times R_{im}$.

```
Procedure preprocess
  begin For $j := 1$ to $m$ do
              Find a non-empty workspace with maximum number of
              suffices including $ij$;
        Eliminate duplications if exist among the obtained workspaces;
        Divide the obtained workspaces into groups so that every
        workspace in each group has at least one suffix common to that
        of another workspace in the same group but no suffices common
        to those in any other group;
```

For example, if we have obtained ws[1, 2], ws[2, 3], and ws[4], they are divided into two groups, one composed of ws[1, 2] and ws[2, 3] and the other composed of ws[4].

Let $G_1, G_2, \ldots, G_{m'}$ be the workspace groups thus far obtained.

For the further process, we need m' temporary areas $s[1], s[2], \ldots, s[m']$ each accommodating a name of intermediate result and the number of element in it, and m' temporary areas $t[1], t[2], \ldots, t[m']$ each accommodating a set of suffices. They must be defined as local variables in the search procedure. We may reserve n areas for each of the above two kinds of temporary areas.

```
      For k := 1 to m' do
begin Renumber the workspaces belonging to $G_k$ in the ascending
         sequence of the number of elements in the intermediate results
         they direct;
```

This is a sort of run-time optimization. We will process the workspaces with a smaller number of elements in the intermediate result it directs before processing the workspaces with a larger number of elements in the intermediate result it directs. By this means, we may minimize the time required in a repeated application of the extended intersection. For example, assume that we have to make $i(i(S_1, S_2), S_3)$. It is obvious that $i(S_1, i(S_2, S_3))$ gives the same result. Let S_1, S_2, and S_3, respectively, contain r_1, r_2, and r_3 elements. The estimated time of the former procedure is

$$f(r_1 \log r_1 + r_2 \log r_2 + r_3 \log r_3 + r \log r) + g(r_1 + r_2 + r_3 + r)$$

where r is the number of elements in $i(S_1, S_2)$, while that of the latter is

$$f(r_1 \log r_1 + r_2 \log r_2 + r_3 \log r_3 + r' \log r') + g(r_1 + r_2 + r_3 + r')$$

where r' is the number of elements in $i(S_2, S_3)$. Hence the former procedure is better if $r < r'$, while the latter is better if $r > r'$. The above arrangement of the workspaces is a heuristics that may achive the optimal execution.

Let $\mathrm{ws}_1, \mathrm{ws}_2, \ldots, \mathrm{ws}_{m''}$ be the workspaces arranged in such a sequence.

```
      Store the content of $\mathrm{ws}_1$ into $s(1)$;
      Store the suffices of $\mathrm{ws}_1$ into $t[k]$;
      For l := 2 to m'' do
begin Execute $i(s[k]\uparrow, \mathrm{ws}_l\uparrow)$;
         Invoke procedure "eliminate";
         Store the name given to the result and the number of elements in
         it into $s[k]$;
```

```
            Store all the distinct suffices in $t[k]$ and those specifying $ws_u$ into
            $t[k]$
        end
    end;
```

Here $x\uparrow$ should be read as "the intermediate result specified by the content of x." The eliminate procedure, which is defined later, is necessary to avoid redundant search process that might be executed when the intermediate result becomes empty.

```
        Renumber the suffix $k$ of $s[k]$ and $t[k]$ in accordance with the
        ascending sequence of the number of elements in $s[k]\uparrow$
    end preprocess;
```

The last statement is again a sort of run-time optimisation to achieve an optimal execution of repeated extended intersection.

For example, if we obtained two group $G_1 = \{ws[1,2], ws[2,3]\}$, and $G_2 = \{ws[4]\}$, the result of the preprocess procedure is

$s[1]$: directs the result of $i(ws[1,2]\uparrow, ws[2,3]\uparrow)$
$t[1]$: contains suffices 1, 2, and 3
$s[2]$: contains the content of ws[4]
$t[2]$: contains suffix 4,

if the number of elements in $i(ws[1,2]\uparrow, ws[2,3]\uparrow)$ is smaller than that in $ws[4]\uparrow$. (Fig. 3). Otherwise $s[1]$ and $s[2]$ as well as $t[1]$ and $t[2]$ are interchanged.

If the given condition is defined on a single relation, m' is always equal to 1. Only $s[1]$ and $t[1]$ is significant in this case.

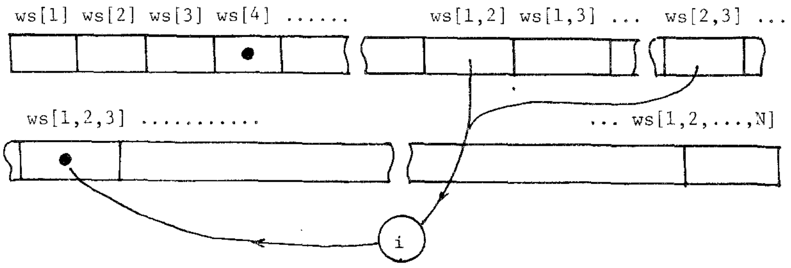

ws[1] ws[2] ws[3] ws[4] ......
ws[1,2] ws[1,3] ... ws[2,3] ...
ws[1,2,3] ....
... ws[1,2,...,N]
i

Fig. 3. Preprocessing for a component condition defined on $R_1 \times R_3 \times R_4$.

5.2. The Conditionprocess Procedure

The conditionprocess procedure is the central part of the search procedure. It processes the given component condition in the most appropriate way according to its type.

```
procedure conditionprocess
  begin If $\mu$ is type $A$ then
    begin Execute $j[\mu](R_{i1})$;
          Invoke procedure "eliminate";
          Store the name given to the result and the number of elements
          in it into $s[0]$;
          If $s[1]$ contains a name other than $R_{i1}$ then
      begin Execute $i(s[0]\uparrow, s[1]\uparrow)$;
            Invoke procedure "eliminate";
            Store the name given to the result and the number of
            elements in it into $s[0]$
      end;
          Store the content of $s[0]$ into the workspace specified by the
          suffices in $t[1]$
    end
```

Here $s[0]$ is a temporary area that accommodates a name of intermediate result and the number of elements in it. It must be defined in the search procedure as a local variable (Fig. 4).

An alternative way to process a type A condition is executing $r[\mu](s[1]\uparrow)$. However this must be executed by a seek on $s[1]\uparrow$. Unlike the extended intersection, the primary key part is not sufficient for the seek and the body relations must be referenced. Hence the above strategy is better in most cases. (Which the better strategy is depends on the implementation-dependent constants a, b, b', e, f, and g and also m and n in Table I. The above is a heuristics.)

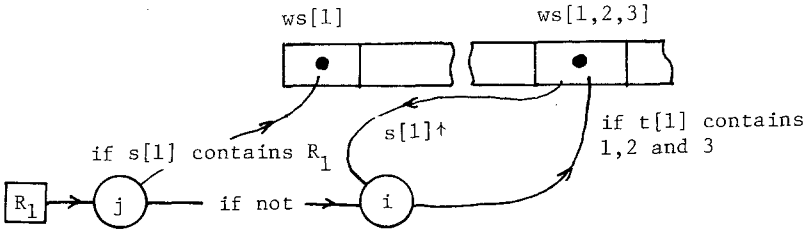


Fig. 4. Processing a type A condition defined on R_1.

```
        Else if $\mu$ is type $B$ then
    begin If $t[1]$ contains only one suffix then
      begin If $\mu$ is preceded by a "$\sim$" operator then
              Execute $j[\sim\mu](s[1]\uparrow)$
            Else
              Execute $j[\mu](s[1]\uparrow)$;
            Invoke procedure "eliminate";
      end
            Else
      begin If $\mu$ is preceded by a "$\sim$" operator then
              Execute $j[\sim\mu](p[i1](s[1]\uparrow)$
            Else
              Execute $j[\mu](p[i1](s[1]\uparrow)$;
            Invoke procedure "eliminate";
            Store the name given to the result and the number of
            elements in it into $s[0]$;
            Execute $i(s[0]\uparrow, s[1]\uparrow)$
      end;
          Store the name given to the result and the number of elements
          in it into the workspace specified by the suffices in $t[1]$
    end
```

Note that the extended join and extended selection are the same operation for the type B condition, which must be carried out by a seek operation (Fig. 5).

Again we use a heuristics in the last half of the above procedure. Instead of applying $r[\mu](s[1]\uparrow)$, we apply $i(j[\mu](p[i1](s[1]\uparrow), s[1]\uparrow))$. In this way, references of the body relations except that corresponding to $p[i1](s[1]\uparrow)$ become unnecessary.

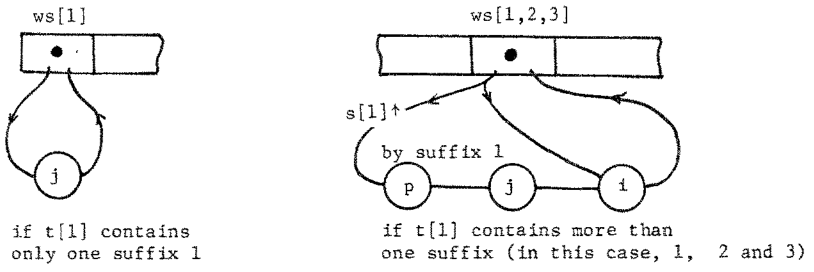

ws[1]
j
if t[1] contains only one suffix 1
ws[1,2,3]
s[1]↑
by suffix 1
p
j
i
if t[1] contains more than one suffix (in this case, 1, 2 and 3)

Fig. 5. Processing a type B condition defined on R_1.

```
        Else if $\mu$ is type $C$ then
    begin For $j := 1$ to $m$ do
      begin Find $t[k]$ which contains $ij$;
              If $t[k]$ contains only one suffix $ij$ then
                Store the name in $s[k]$ into $u[j]$
              Else
        begin Execute $p[ij](s[k]\uparrow)$;
                Store the name given to the result into $u[j]$
        end
      end;
          Execute $j[\mu](u[1]\uparrow, u[2]\uparrow, ..., u[m]\uparrow)$;
          Invoke procedure "eliminate";
          Store the name given to the result and the number of elements
          in it into $s[0]$;
          For $k := 1$ to $m'$ do
      begin If $t[k]$ contains more than one suffix then
        begin Execute $i(s[0]\uparrow, s[k]\uparrow)$;
                Store the name given to the result and the number of
                elements in it into $s[0]$
        end
      end;
          Store the content of $s[0]$ into the workspace specified by all
          the suffices in $t[k]$'s
    end
```

Here $u[j]$s are another set of temporary areas that accommodate names of intermediate results. They must be defined in the search procedure as local variables.

The $j[\mu](u[1]\uparrow, u[2]\uparrow, u[2]\uparrow, ..., u[m]\uparrow)$ operation can be achieved by a sequential collation of index files, which are already provided or temporarily created. This procedure can deal with partial matches by introducing imaginary tuples.(19) It will be obvious that the above procedure is much faster than dealing with Cartesian product files.

The last statement uses the union of the contents in $t[k]$s $(1 \leqslant k \leqslant m')$. For example, if $t[1]$ contains 1, 2, 3, and $t[2]$ contains 4, the content of $s[0]$ is stored into ws[1, 2, 3, 4] (Fig. 6).

```
        Else
    begin For $k := 1$ to $m'$ do
      begin Find the suffices in $t[k]$ which are included in the set
              composed of $i1, i2, ..., im$;
```

This is the statement obtaining the intersection of the set of suffices in $t[k]$ and $\{i1, i2, ..., im\}$. Let $h1, h2, ..., hm'''$ be such suffices.

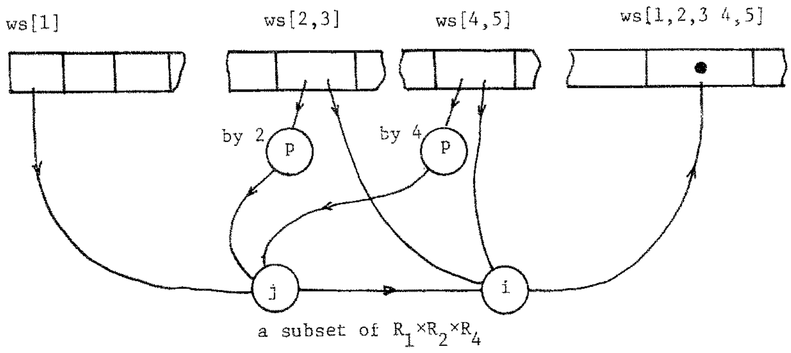


Fig. 6. Processing a type C condition defined on $R_1 \times R_2 \times R_4$.

```
    If $t[k]$ contains no more suffices than $h1, h2,..., hm'''$ then
begin Store the name in $s[k]$ into $u[k]$;
      Clear $s[k]$
end
   Else
begin Execute $p[h1,h2,..., hm'''](s[k]\uparrow)$;
      Store the name given to the result into $u[k]$
end
end;
    Store the name in $u[1]$ into $s[0]$;
    For $k := 2$ to $m'$ do
begin Execute $i(s[0]\uparrow, u[k]\uparrow)$;
```

This extended intersection implies making a Cartesian product.

```
      Store the name given to the result into $s[0]$
end;
  If $\mu$ is preceded by a "$\sim$" operator then
     Execute $r[\sim\mu](s[0]\uparrow)$
  Else
     Execute $r[\mu](s[0]\uparrow)$;
  Invoke procedure "eliminate";
  Store the name given to the result and the number of elements in
  it into $s[0]$;
  For $k := 1$ to $m'$ do
     If $s[k]$ is non-empty
begin Execute $i(s[0]\uparrow, s[k]\uparrow)$;
```

```
                Store the name given to the result and the number of elements
                in it into $s[0]$
            *end*;
                Store the content of $s[0]$ into the workspace specified by all
                the suffices in $t[k]$'s
        *end*
    *end* conditionprocess;
```

The above procedure for type D conditions is very time-consuming because it includes the two most time-consuming elementary operations, extended intersection making a Cartesian product and extended selection for a subset of Cartesian product of relations. The user may be given a warning message when such a procedure is invoked (Fig. 7).

5.3. The Eliminate Procedure

The eliminate procedure is necessary to discontinue the further search processes when the result becomes an empty set.

```
procedure eliminate
  begin If the result becomes empty then
    begin Eliminate all the conditions and the compound conditions
          enclosed in parentheses, that are conjunctively combined with
          the condition currently having been processed;
          Exit to the statement to which label $\Delta$ is given
    end
  end eliminate;
```

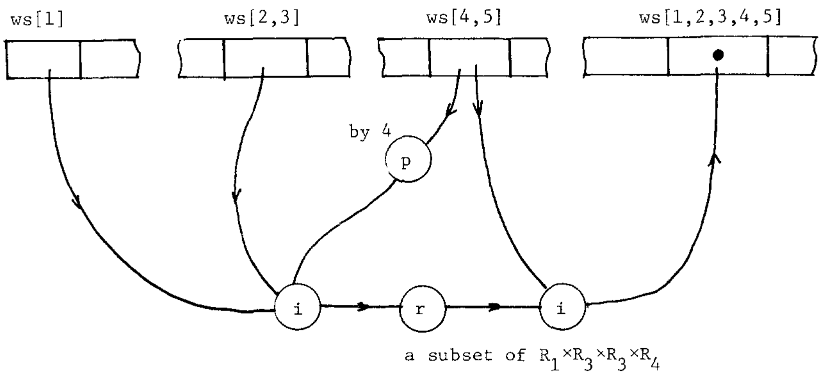


Fig. 7. Processing a type D condition defined on $R_1 \times R_2 \times R_3 \times R_4$.

By returning to Δ point directly, the contents of workspaces currently being used are eliminated.

5.4. The Postprocess Procedure

As mentioned in Sec. 2, the procedures described in Sec. 7.2 dealing with various types of component conditions are efficient only when the ratio of the number q of qualified elements to the number p of the elements to be examined is considerably small. If the obtained intermediate result contains a small number of elements than a certain value, say ω, it is better to process the further search collectively by a seek operation. The postprocess procedure enables such a run-time optimization.

Let us assume that the result of the conditionprocess procedure has been stored in ws[$g1, g2, \ldots, gm''''$].

```
procedure postprocess
  begin If the result contains less than $\omega$ elements then
    begin Change the type of all the type $A$ conditions, which are
          conjunctively combined with $\mu$ and are defined on one of
          $R_{g1}, R_{g2}, \ldots, R_{gm''''}$, to type $B$;
          Change the type of all the type $C$ conditions, which are
          conjunctively combined with $\mu$ and are defined on a Cartesian
          product of some of $R_{g1}, R_{g2}, \ldots, R_{gm''''}$, to type $D$;
          If there is a type $B$ condition, which is conjunctively combined
          with the type $B$ condition obtained as above and defined on the
          relation on which the above obtained type $B$ condition is
          defined, this conjunct is redefined as a type $B$ condition;
          If there are conditions, which are conjunctively combined with
          the type $D$ condition obtained as above and defined on the
          Cartesian product of some of relations that compose the
          Cartesian product on which the above obtained type $D$
          condition is defined, this conjunct is redefined as a type $D$
          condition
    end
  end postprocess;
```

The ω value is implementation-dependent. Two different values can be used for the case in which the result of the conditionprocess procedure is a subset of a single relation and the case in which it is a subset of a Cartesian product of more than one relation. They may be determined using implementation-dependent constants appeared in Table I or by the instrumentation in actual search operations.

5.5. The Union Procedure

When all the component conditions have been processed, the result is stored in the X stack. If an "V" operator had appeared during the search execution, an additional entry was pushed down in the X stack. Therefore, we have to make union of the intermediate results directed by the X stack entries. The union procedure uses the Z stack.

```
procedure union
  begin Store the empty result symbol $\phi$ into ws[1, 2,..., $n$];
      $\nabla$ While there remain some entries in the $X$ stack to be popped
          up do
    begin Pop up the $X$ stack into the workspaces;
          For $k := n$ downto 2 do
      begin While there are non-empty workspaces with $k$ suffices do
        begin Pick up a non-empty workspace with $k$ suffices;
              Clear the workspace all whose suffices are included in
              the above $k$ suffices
        end;
          Renumber all the non-empty workspaces in the ascending
          sequence of the number of elements in the intermediate results
          they direct;
```

This is again a sort of run-time optimization to minimize the time required in a repeated application of the extended intersection. Let ws_1, ws_2,..., ws_m be the workspaces arranged in such a sequence.

```
          Store the content of $\mathrm{ws}_1$ into $s[0]$;
          For $k := 2$ to $m$ do
        begin Execute $i(s[0]\uparrow, \mathrm{ws}_k\uparrow)$;
              If the result becomes empty then
                 Exist to the statement to which label $\nabla$ is given;
              Store the name given to the result and the number of
              elements in it into $s[0]$
        end;
            Push down $s[0]$ into the $Z$ stack
      end;
```

Now each entry in the Z stack contains a name of intermediate result, which is a subset of the Cartesian product of n relations on which the given condition is defined.

```
          If $Z$ stack is not empty then
        begin Pop up the $Z$ stack into ws[1, 2,..., $n$];
              While there remain some entries in the $Z$ stack to be
              popped up do
```

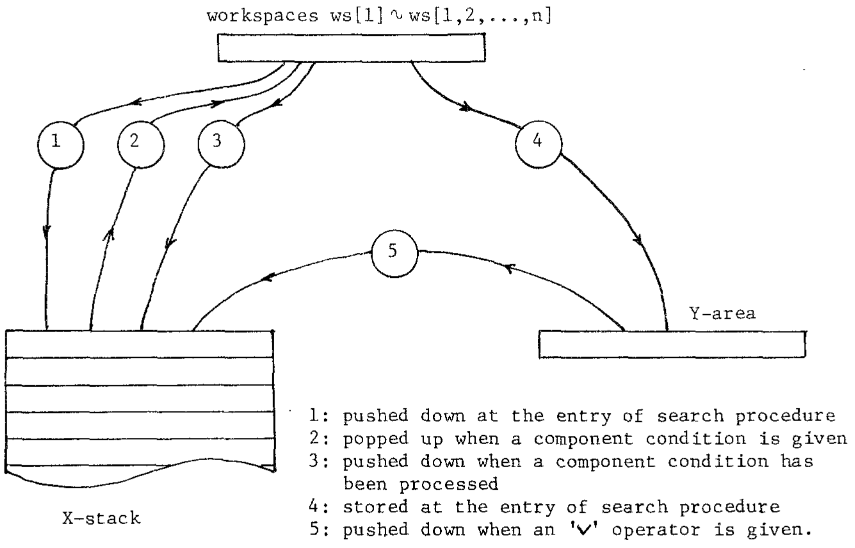

workspaces ws[1] ∿ ws[1,2,...,n]
1
2
3
4
5
Y-area
X-stack
1: pushed down at the entry of search procedure
2: popped up when a component condition is given
3: pushed down when a component condition has been processed
4: stored at the entry of search procedure
5: pushed down when an '∨' operator is given.

Fig. 8. Uses of the *X* stack and the *Y* area in the recursive search procedure.

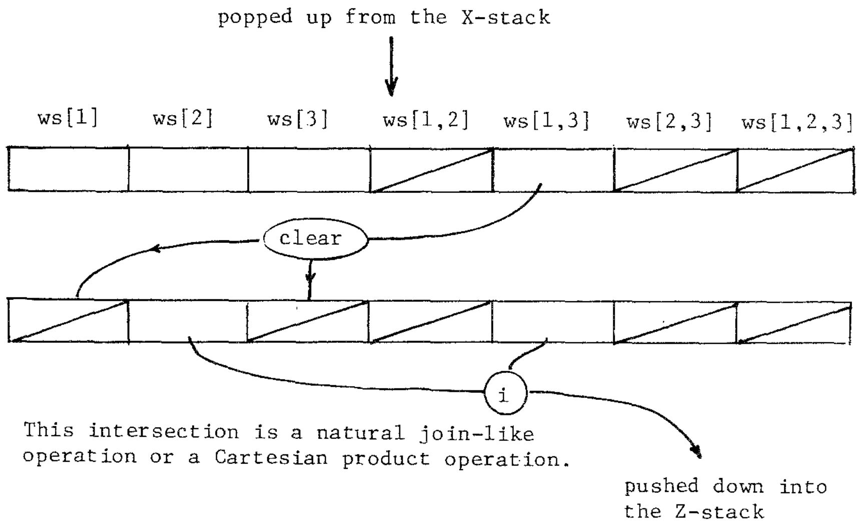

popped up from the X-stack
ws[1]
ws[2]
ws[3]
ws[1,2]
ws[1,3]
ws[2,3]
ws[1,2,3]
clear
i
This intersection is a natural join-like operation or a Cartesian product operation.
pushed down into the Z-stack

Fig. 9. Preparation for making the union of intermediate results.

```
        begin Pop up the $Z$ stack into $s[0]$;
              Execute $u(\mathrm{ws}[1, 2,\dots, n]\uparrow, s[0]\uparrow)$;
              Store the name given to the result and the number of
              elements in it into $\mathrm{ws}[1, 2,\dots, n]$
        end
    end
end union;
```

Figure 8 shows the use of the X stack and the Y area. Figure 9 shows the first half of the union procedure, that create an intermediate result to be pushed down into the Z stack. The last half makes the union of the intermediate results directed by the Z stack entries.

6. SEARCH EXECUTION

Actual search can be achieved by executing the following six steps.

STEP 11: Provision of the workspaces. We provide $2^N - 1$ workspaces each being able to accommodate the name of a subset of a Cartesian product of each combination of operand relations.

The workspaces are identified as shown in Table II.

STEP 12: Invocation of the recursive search procedure. We first place the name and the number of tuples (if known) of each operand relation into the corresponding workspace ws$[i]$, *and clear all other workspaces (with more than one suffix). Then we eliminate the outermost parentheses in the given syntactically transformed condition and invoke the recursive search procedure with $n = N$, λ being the given condition and the above workspaces as actual parameters.*

This step terminates with the name of the result and the number of elements in it in ws$[1, 2,\dots, N]$.

STEP 13: Processing quantifications. We examine the quantification part of the given condition from right to left.

(1) If Γ_k is a universal quantification, then we execute

$$v(\mathrm{ws}[1, 2,\dots, N]\uparrow, R_k)$$

and store the name given to the result and the number of elements in it back into ws$[1, 2,\dots, N]$.

(2) If Γ_k is an existential quantification, then we find all the existential quantifications $\Gamma_{k-p}, \Gamma_{k-p+1}, \ldots, \Gamma_{k-1}$ immediately preceding Γ_k, execute

$$p[1, 2, \ldots, k-p-1](\mathrm{ws}[1, 2, \ldots, N]\uparrow),$$

and store the name given to the result and the number of elements in it back into ws$[1, 2, \ldots, N]$.

This step is repeated until all the quantifications have been processed.

Note the sorting operation for the ws$[1, 2, \ldots, N]$ is necessary only for the first division or projection.

Now the ws$[1, 2, \ldots, N]$ directs the final result which is a set of primary keys or a set of ordered set of primary keys of qualified elements.

STEP 14: Processing the artificially added free variable. If R_1 is the relation added in step 1 *as the domain of an artificial free variable, the answer is "yes" if the result is not empty. Otherwise the answer is "no."*

STEP 15: Fetching the result. Fetch the qualified tuples or qualified ordered sets of tuples with reference to the search result directed by the content of ws$[1, 2, \ldots, N]$.

Either the tuplewise (piped mode) read or the set (non-piped mode) read is employed in this step. If necessary, we can answer the question that asks the number of qualified elements.

STEP 16: Releasing the storage areas. We release all the areas that accommodate the result and the intermediate results, that are not released as yet.

7. CONCLUDING REMARKS

In Sec. 1.3, we pointed out four major problems regarding the preceding works. Here, we will review how these problems have been solved in our algorithm.

1. Our algorithm can deal with the extended relational calculus. The search condition can be arbitrary logical function defined on a Cartesian product of relations. Some examples that are not relational calculi but extended relational calculi were shown in Sec. 1.2.

2. Instead of using relational algebra operations, we use our elementary set operations, which resemble the relational algebra operations but somewhat extended. The major difference between the two is our operations keep only the primary key part of the intermediate result as mentioned in Sec. 4.

3. The target list specification is separated from our search algorithm. It is completely left to the care of the host language program. This enables any complicated computation to be applied to the search result.

4. Type *C* conditions are processed by a sequential collation of *n* relations (or sometimes a sequential collation applied to *n* index files). Such type *C* conditions very frequently appear in actual applications.

In consequence, we can expect a much wider application of the search algorithm than those based on the relational calculus and relational algebra.

The algorithm presented in this paper does not achieve the optimality in a precise sense because several heuristic techniques are integrated in the algorithm to provide a certain suboptimal algorithm. The optimal algorithm can only be established with many implementation-dependent factors taken into account.

Up to this date, a small portion of the algorithm (for the search condition defined on a single relation) has been implemented in FORIMS.(2,8)

REFERENCES

1. M. M. Astrahan and D. D. Chamberlin, "Implementation of the Structured English Query Language," *Comm. ACM*, **18**(10), 580–588 (1975).
2. Y. Chiba, "A Data Base Search Algorithm Based on Complicated Retrieval Algorithms," *The Soken Kiyo*, **5**(1), Nippon Univac Sogo Kenkyusho, Inc., pp. 159–176 (1975).
3. E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus," In *Proceedings ACM SIGMOD '71 Workshop on Data Description, Access and Control*, 1971, pp. 35–68.
4. E. F. Cood, "Relational Completeness of Data Base Sublanguages," in *Data Base Systems*, Courant Computer Science Symposium, 6, R. Rustin, ed. (Prentice-Hall, Englewood Cliffs, New Jersey, 1972), pp. 65–98.
5. G. D. Held, M. R. Stonebraker, and E. Wong, "INGRES—A Relational Data Base System," In *Proceedings of AFIPS '75 NCC*, (1975), pp. 409–416.
6. I. Kobayashi, "An Overview of Database Management Technology," Sanno College of Management and Informatics, TRCS-4, also to appear in *Advances in Information Systems Science*, 9, J. T. Tou, ed., (Plenum Press, New York, 1982).
7. I. Kobayashi, "Manipulating Database Relations," Sanno College of Management and Informatics, TRCS-5, 1980.
8. K. Kohri and Y. Chiba, "FORIMS Phase 2 Design Specification: A FORTRAN Oriented Information Management System," *The Soken Kiyo*, **5**(10), Nippon Univac Sogo Kenkyusho, Inc. (1975), pp. 177–210.
9. D. E. Knuth, *The Art of Computer Programming 3*, *Sorting and Searching* (Addison-Wesley, Reading, Massachusetts, 1968).
10. J. Martin, *Principles of Data-Base Management* (Prentice-Hall, Englewood Cliffs, New Jersey, 1976).
11. F. P. Palermo, "*A Data Base Search Problem*," In *Proceedings of the 4th International Symposium Computer Information Science* (Plenum Press, New York, 1972), pp. 67–101.

12. R. Reiter, "Query Optimization for Question-Answering Systems," in *Proceedings of the COLING Conference*, Ottawa (1976).
13. J. B. Rothnie, "Evaluating Inter-Entry Retrieval Expressions in a Relational Database Management System," in *Proceedings of the AFIPS '75 NCC* (1975), pp. 417–423.
14. J. M. Smith and P. Y. T. Chang, "Optimizing the Performance of Relational Algebra Database Interface," *Comm. ACM*, **18**(10), 568–588 (1975).
15. G. Salton and A. Wong, "Generation and Search of Clustered Files," *ACM Trans. Database Sys.* **3**(4), 321–346 (1978).
16. S. Todd, "PRTV: An Efficient Implementation of Large Relational Data Bases," in *Proceedings of the 1st International Conference on Very Large Data Bases* (1975), pp. 544–556.
17. J. T. Tou, "Design of Medical Knowledge System for Diagnostic Consultation and Clinical Decision-Making," in *Proceedings of the International Computer Symposium '78* (1978), pp. 80–99.
18. G. Wiederhold, *Database Design* (McGraw-Hill, New York, 1977).
19. H. K. T. Wong and K. Youssefi, "Decomposition: A Strategy for Query Processing," *ACM Trans. Database Syst.* **1**(3), 233–241 (1976).
20. S. B. Yao, "Optimization of Query Evaluation Algorithms," *ACM Trans. Database Syst.* **4**(2), 133–155 (1979).