Management/ Database Systems

H. Morgan Editor

# Implementation of a Structured English Query Language

M.M. Astrahan and D.D. Chamberlin
IBM Research Division, San Jose

**The relational model of data, the XRM Relational Memory System, and the SEQUEL language have been covered in previous papers and are reviewed. SEQUEL is a relational data sublanguage intended for ad hoc interactive problem solving by non-computer specialists. A version of SEQUEL that has been implemented in a prototype interpreter is described. The interpreter is designed to minimize the data accessing operations required to respond to an arbitrary query. The optimization algorithms designed for this purpose are described.**

**Key Words and Phrases: relational model, query language, nonprocedural language, database, data structure, data organization**

**CR Categories: 3.74, 3.75, 4.22, 4.33, 4.34**

A version of this paper was presented at the ACM SIGMOD Conference on the Management of Data, San Jose, Calif., May 14-16, 1975.

Authors' address: IBM Research Division, Monterey & Cottle Roads, San Jose, CA 95193.

## Introduction

In a series of papers, Codd [7-9] has introduced the relational model of data and discussed its advantages in terms of simplicity, symmetry, data independence, and semantic completeness. The relational model suggests that all data in a database management system may be represented by a set of named tables, or "relations." Within each relation, the columns are named, and the ordering of rows is considered immaterial. Each row describes one instance of the concept or entity described by the relation. The rows must be distinct; duplicates are not allowed. Each relation has a column or set of columns, called the primary key, whose values must be unique in the relation. An example of a relation describing the employees of a company is shown in Figure 1. The primary key of this relation is MNO.

A number of languages have been proposed for expressing queries against a relational database [2, 4, 5, 9, 10]. These languages fall into two major categories: those based on relational algebra and those based on relational calculus. Both the relational algebra and the relational calculus were originally proposed by Codd [8]. The relational algebra is based on a series of high level procedural operators such as "join" and "projection." The relational calculus is a nonprocedural language based on the mathematical quantifiers "for all" ($\forall$) and "there exists" ($\exists$).

SEQUEL, the language discussed in this paper, is a relational query language based on neither relational algebra nor relational calculus. SEQUEL is a nonprocedural language which does not make use of quantifiers or other mathematical concepts; rather, SEQUEL uses a block structured format of English key words (hence the acronym "*S*tructured *E*nglish *Q*uery *L*anguage"). SEQUEL is intended for interactive, problem solving use by people who have need for interacaction with a large database but who are not trained programmers. This class of users includes urban planners, sociologists, accountants, and other professionals. The objective of the language is to provide a simple, easy-to-learn means of expressing the primitive actions used by people to obtain information from tables, such as "look up a value in a column." SEQUEL and its companion language, SQUARE, have been shown to be relationally complete, i.e. equivalent in power to Codd's relational calculus [2, 5, 8].

This paper describes a prototype system, now running at the IBM Research Laboratory in San Jose, California, which implements a version of SEQUEL in the environment of XRM, a relational memory system. The goals of the project were: (a) to gain operational experience in using the SEQUEL language; (b) to develop and test algorithms for optimizing retrieval operations in response to an ad hoc query; (c) to gain experience and evaluate the use of the XRM interface; (d) to demonstrate the power of the SEQUEL language; and (e) to furnish a test bed for the evaluation of new functions.

Fig. 1.

EMP:

| MNO | NAME | DNO | MGR | TITLE | SAL |
|---|---|---|---|---|---|
| 101 | ABEL | 19 | 510 | DIRECTOR | 18000 |
| 102 | BAKER | 19 | 525 | DIRECTOR | 19000 |
| 103 | CARLSON | 19 | 525 | CLERK | 8000 |
| 104 | DOE | 19 | 525 | VICE-PRES | 22000 |
| 105 | EASTMAN | 20 | 525 | CHAIRMAN | 24000 |

The SEQUEL Interpreter prototype was designed for rapid implementation by a small group. The design and implementation were done in eight months. The part-time efforts of four people were involved, amounting to about 22 man-months. A fifth person provided part-time support in correcting XRM bugs.

The version of SEQUEL implemented by the prototype includes facilities for query, insertion, deletion, and update of relations, as well as the ability to create new relations and to control the set of indexes maintained on the database. The query features of this version of SEQUEL are described in the next section. The following sections review the key features of XRM and describe the organization and algorithms of the SEQUEL interpreter. Finally, the directions of future work are summarized.

## The Sequel Language

The query facilities of SEQUEL will be introduced by a series of example queries against a database which describes a small company. The database contains the following tables:

```
EMP (MNO, NAME, DNO, MGR, TITLE, SAL)
DEPT (DNO, DNAME, LOC)
```

The EMP table gives each employee's man number, name, title, salary, the man number of his manager, and his department number. The DEPT table gives, for each department, its number, name, and the city in which it is located.

The simplest operation in SEQUEL is called a mapping, and corresponds to the act of finding the value in a table which is associated with another, known value. The basic mapping is illustrated by Q1.

Q1. Find the department numbers of departments which are located in Newberg.

```
SELECT DNO
FROM   DEPT
WHERE LOC = 'NEWBERG';
```

The mapping consists of three keywords: SELECT, FROM, and WHERE; and three parameters: the table to which the query is directed (EMP), the names of the items to be returned (DNO), and the condition to be satisfied (LOC = 'NEWBERG'). This simple format is called a query block and is used throughout the SEQUEL language.

SEQUEL allows the basic mapping to be extended in several ways. A query may specify more than one item to be selected, and may specify a complex boolean condition in the WHERE clause, as illustrated by Q2.

Q2. List the name and manager of all employees in department 19 who earn more than 15000.

```
SELECT NAME, MGR
FROM   EMP
WHERE DNO = 19
AND     SAL >15000;
```

If the WHERE clause is omitted from the query block, the query returns all values from the selected column of the table. Duplicate values are deleted from the list. This feature, illustrated by Q3, corresponds to Codd's "projection" operator.

Q3. List all the different department numbers in the employee table.

```
SELECT DNO
FROM   EMP;
```

If the user wishes to select the entire row whenever the WHERE-condition is satisfied, he may use the abbreviation SELECT * . This feature, illustrated by Q4, corresponds to Codd's "restriction" operator.

Q4. List the EMP rows for those employees who earn more than 20000.

```
SELECT *
FROM   EMP
WHERE SAL > 20000;
```

SEQUEL allows the user to specify any of several built-in functions in the SELECT clause of a query block. These functions include SUM, COUNT, AVG, MAX, and MIN. The function is performed over the set of rows which satisfy the WHERE clause. For example:

Q5. Find the average salary of employees in department 17.

```
SELECT AVG(SAL)
FROM   EMP
WHERE DNO = 17;
```

The WHERE clause may be used either to compare a value with a constant (as above) or to test a value for inclusion in a set, as illustrated by Q6. Set inclusion is denoted by the word IN.

Q6. List the names of employees who are in department 17, 24, or 32.

```
SELECT NAME
FROM   EMP
WHERE DNO IN (17, 24, 32);
```

The ability to test for set inclusion in a WHERE clause enables the user to nest query blocks inside each other, using the output of one mapping as the input to another, as illustred by Q7. (For syntactic purposes, each query block is terminated by a semicolon. Since one block is nested inside another in this example, two semicolons are required.)

Q7. List the names of employees who work for departments which are located in Newberg.

```
SELECT NAME
FROM   EMP
WHERE DNO IN
        SELECT DNO
        FROM   DEPT
        WHERE LOC = 'NEWBERG';;
```

In this query, the inner mapping finds the set of department numbers of all those departments in Newberg; the outer mapping then returns the names of those employees whose department number matches any number in this set. In this way, query blocks may be nested inside each other to many levels of nesting.

It is occasionally necessary in constructing a SEQUEL query to indicate a correlation between two query blocks. For this purpose, SEQUEL allows a correlation variable to be placed in the FROM clause of a query block. The correlation variable represents a row of the indicated table, and may be used in other query blocks to refer back to this row. For example:

Q8. Find names of employees who earn more than their manager.

```
SELECT NAME
FROM   E IN EMP
WHERE  SAL >
       SELECT SAL
       FROM   EMP
       WHERE  MNO = E.MGR;;
```

The outer block returns the name of employee E whenever E's salary is greater than the value returned by the inner block. The inner block finds the salary of E's manager by a simple mapping, referring to E by means of the correlation variable.

The usefulness of the correlation variable is further illustrated by Q9, which also shows how the * abbreviation (meaning "entire row") may be used with a built-in function:

Q9. List the department numbers and names of departments having more than ten employees.

```
SELECT DNO, DNAME
FROM   D IN DEPT
WHERE  10 <
       SELECT COUNT (*)
       FROM   EMP
       WHERE  DNO = D.DNO;;
```

Some queries call for information to be returned which cannot be obtained by simply selecting items from a table or using a built-in function. For example, a user may ask for a list of employee names together with the names of the departments in which they work. This information is not all present in a single table; the employee names are in the EMP table and the department names are in the DEPT table. The correlation between the two must be done by means of the column DNO, which appears in both tables. When the information needed by a query cannot be obtained by simple selection, SEQUEL allows the user to place an arbitrary variable called a "computed variable" in the SELECT clause. A special COMPUTE clause is then used to give the definition of the computed variable, using a nested query block. The solution to the above example is shown in Q10.

Q10. List employee names together with the names of the departments in which they work.

```
SELECT  NAME,Q
FROM    E IN EMP
COMPUTE Q =
        SELECT DNAME
        FROM   DEPT
        WHERE  DNO = E.DNO;;
```

The computed variable used together with built-in functions gives a powerful query capability, as illustrated by Q11.

Q11. List for each department its department number, name, and the average salary of its employees.

```
SELECT  DNO, DNAME, Q
FROM    D IN DEPT
COMPUTE Q =
        SELECT AVG(SAL)
        FROM   EMP
        WHERE  DNO = D.DNO;;
```

## XRM Relational Memory System

The SEQUEL Interpreter relies upon the XRM (Extended n-ary Relational Memory [12]) system for storage and retrieval of data in the form of n-ary relations. XRM was developed at the IBM Cambridge Scientific Center. XRM in turn is based upon the Cambridge RM (Relational Memory) system [13] which provides: (1) storage of variable length byte strings (entities) addressed by numerical identifiers, and (2) efficient storage and retrieval of numerical binary relations (sets of pairs of integers).

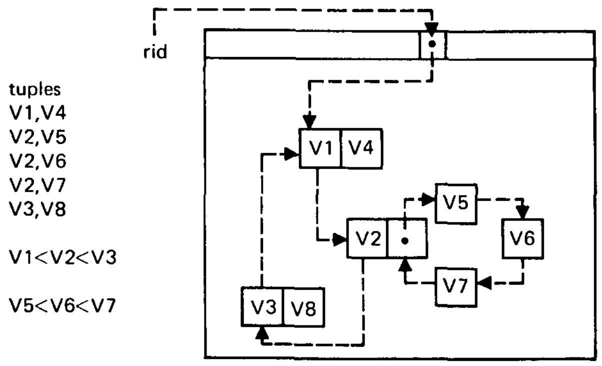
RM is a single-user system. It is low level in the sense that the interface with a calling program is in terms of internal identifiers instead of external names and that parameters for controlling physical layout are available at the interface.

Data in RM is stored in pages of 4096 bytes. One set of pages is used to store the entities. A data identifier for an entity is made up of a page address and an offset in a page header area. The identified header position contains a pointer to the location of the data on the page. Another set of pages is used to store the binary relations. The identifier of a relation is made up of a page address and an offset in a page header area which contains a pointer to the first tuple (pair). On an RM relation page, the first elements of the tuples of a relation are connected by a chain of pointers and are accessed in order of increasing magnitude, as shown in Figure 2.

All tuples which contain the same first element value are represented by an ordered chain of second element values connected to the first element value. The actual physical locations of the first element values on the page are determined by a hash coding technique which allows rapid access to a tuple with a given first element value. For large relations, elaborate overflow techniques allow access to any tuple with at most two page accesses. RM provides for sequential retrieval of all pairs of a binary relation or of those with a given first element value, as well as an efficient test for the existence of a given pair.

The identifiers are automatically allocated by RM. They constitute an address space. RM supports multiple address spaces with independent SAVE and RESTORE capabilities. This is very useful for implementing workspaces [14].

Fig. 2. RM relation page.



XRM is built upon binary RM. Each tuple of an n-ary relation is stored as a byte string in RM "entity" space. The string contains a fixed length integer for each column of the relation. This number is an actual value if the domain is composed of integers. If not, it is generally the identifier of another entity, containing the value. Those entities corresponding to the same domain constitute a "class" relation. A class relation provides for quick access from the value to the identifier. It is implemented by using a binary relation whose first element is a hash function of the value and whose second element is the identifier of the stored value. Since hash collisions are possible, the actual value must be retrieved for verification.

A tuple can be rapidly retrieved by means of its identifier, now called a tuple identifier, or tid. As in Codd's Gamma-0 [1], there is a master relation which contains information describing each of the other relations. The relation identifier (rid) of a relation is the tid of a tuple in the master relation, and that tuple describes the relation. It gives information about the type, degree, column encoding. inversions etc. Associated with each n-ary relation is a binary relation which permits rapid access to a particular tuple of the n-ary relation given its primary key value. Corresponding to each n-tuple is a tuple of the binary relation whose elements are a hash function of the key value and the tid of the n-tuple. This is similar to the CALC KEY function in DBTG [6]. Rapid access to tuples containing a particular value in a particular column can also be provided by an inversion relation. An inversion is a binary relation whose elements are a column value and a tid. When an inversion is specified for a column, the system automatically updates it after changes to the contents of the n-ary relation.

XRM provides for retrieval of all the tuples of a relation by a sequence of NEXT commands and for retrieval of a tuple given its tid or its primary key value. The tid's corresponding to tuples with a given value in a given column of an n-ary relation may be sequentially retrieved if an inversion relation exists for the column.

## Sequel Interpreter Prototype

The SEQUEL Interpreter implements the SEQUEL language using XRM for physical representation of the relations. The interpreter must translate nonprocedural, set oriented statements into an efficient sequence of tuple-at-a-time XRM commands. To this end, several optimization rules are employed. The basic objective of the optimization is to minimize the response time for a query. In order to do this we try to minimize the number of tuple retrieval operations. We would prefer to have as an objective the minimization of the number of disk accesses, on the assumption that we can retrieve and process many tuples from a page in high speed storage in the time of one disk access to a page. However, two factors make this objective impractical. First, page accessing cannot be directly controlled through the XRM interface, although it can be indirectly influenced by parameters which influence the physical placement of inserted data. Second, in a virtual storage system being shared among many users, the Interpreter cannot control the residency of pages in real high speed storage. The Interpreter algorithms do affect the number of tuple retrieval operations and we have assumed that disk accesses and response time will be correlated with this number.

The SEQUEL Interpreter is a PL/I program. Its main sections are a Parser, an Optimizer, and a Scanner. The Parser is of the LALR($k$) type. It is based upon the work of Lalonde and many others [11]. We have added semantic routines which cause a SEQUEL statement to be transformed into a direct representation of the corresponding parse tree. Each WHERE clause is represented as a binary tree of predicates connected by AND and OR nodes. The Optimizer and the Scanner work with the parse tree to interpret the SEQUEL statement. The function of the Optimizer is to minimize tuple retrieval operations by working as much as possible with lists of tuple tid's retrieved by means of inversion relations. An inversion relation will henceforth be called an index. The Optimizer determines for each query block a scan list consisting of the tid's of only those tuples which might satisfy the predicate tree of the block. Tuples which can be shown by indexes not to satisfy the block are excluded from the scan list. For some predicate combinations the scan list represents precisely those tuples which satisfy the block. In this case, we say the block is "completely resolvable by index." In other cases, the tuples represented by the scan list must be materialized and tested against the actual predicates of the block. This operation is performed by the Scanner. The Scanner also prepares an output list for the block by selecting the desired attributes from the successful tuples or by computing the values of built-in functions.

The interface between the user and the Interpreter is the UFI (User Friendly Interface). The UFI operates the display scope. It accepts the user's SEQUEL statement, transmits it to the Interpreter and displays the results to the user. In general, the result of a query is a table of data, and the UFI presents a "window" on the table, displaying as many rows and columns as will fit on the screen. A typical display resulting from the query SELECT * FROM EMP; looks very much like the example in Figure 1. The "window" can be moved left, right, up, and down under user control to display the whole table.

The Interpreter maintains a set of catalog relations for translation between the user's names for relations and columns and their internal encoding in XRM. Data type (character string or integer) and other information is also kept for the domains on which columns are defined. These catalog relations are updated in response to SEQUEL statements which create and destroy relations and inversions. A relation must be completely defined before any tuples are inserted. In the prototype Interpreter no changes may be made to the definition of a relation after tuples have been inserted.

Most catalog maintenance and database maintenance statements translate rather directly into XRM commands. However, some INSERT, UPDATE, and DELETE statements may contain a query block (e.g. to identify the tuples to be updated.) For these, the Optimizer and Scanner are called to process the query block.

The basic tool for minimizing tuple retrieval operations in interpreting a query is the use of an index (inversion) to retrieve a list of tids of tuples having a given value in a given column of a relation. Indexes may be used with predicates of type "column-name = value" or "column-name IN value-list". Since indexes involve extra overhead during insertions and deletions, they will only be maintained for those columns anticipated to appear in many query predicates. Therefore, for predicates of the right type, the Interpreter must still determine if an index exists. Where indexes can be used, AND and OR combinations of predicates can be evaluated by union and intersection operations on lists of tid's. These operations are more efficient in XRM than the corresponding operations on the tuples. However, an index will not always be used, even though it corresponds to a predicate of the above mentioned types. This is because it may be OR'd with a predicate, such as "column-name > value", which cannot use an index and therefore requires a complete scan of all the tuples. An index will only be used if it reduces the number of tuples which must be retrieved.

The Optimizer algorithms will be described in terms of three steps, called Steps A, B, and C, and three auxiliary algorithms: the Index Selection, List Combining, and Test Tree algorithms.

## Step A

In evaluating a query block Step A of the Optimizer classifies each predicate of the query block into one of five types, based on how the predicate must be resolved. (To "resolve" a predicate means to find the set of tuples which satisfy it.) The five predicate types are:

**Type P1.** Predicates which can be immediately resolved by an existing index. They have the form "column-name = value" or "column-name IN value-list". (An example is DNO = 27 in the EMP relation, if there is an index on DNO.) Such a predicate may not contain a correlation to an outer query block. It may, however, contain a nested query block if the nested query does not involve any correlation to a higher level block. An example is:

```
DNO IN
    SELECT DNO
    FROM   DEPT
    WHERE LOC = 'NEWBERG';
```

Such a nested query will be completely evaluated during the classification of the predicate from the containing block by a recursive call to the Optimizer. The nested query will be in effect replaced by the list of values resulting from the evaluation, as if the predicate had been DNO IN(10, 27, 42). A type P1 predicate will later be, in effect, replaced by a tid-list retrieved from the index. If the predicate contains a value-list, the tid-lists corresponding to each value will be unioned. A special case of type P1 is when the column-name constitutes the primary key of the relation. In this case there will be no index but the single tid corresponding to each value can be efficiently retrieved.

**Type P2.** Predicates which cannot be resolved by an existing index, but which do not involve correlation to a higher level query block. An example is DNO = 27 if there is no index on DNO in EMP. Another example is DNO > 27 even if there is an index on DNO, since an indefinite value-list is involved. A P2 predicate may contain a nested query that does not involve higher level correlation, as for type P1. A P2 predicate requires a scan of the relation in order to determine which tuples satisfy it. However, before the scan is made we will try to reduce the scan list by using P1 predicates, as described below.

**Type P3.** Predicates which contain a correlation to a higher level query block but which can be resolved by an index if a tuple from the higher block is given. An example of a query block containing such a predicate is:

```
SELECT NAME FROM D IN DEPT WHERE 10 <
       SELECT COUNT (*) FROM EMP WHERE DNO =
       D.DNO;;
```

This query finds the names of departments having more than 10 employees. The predicate DNO = D. DNO is type P3 if there is an index on DNO in EMP. Each time a tuple from DEPT is tested during a scan of DEPT, its DNO value will replace D.DNO in the predicate on EMP and the predicate will be resolved into a tid-list by means of the index. As with type P1, a special case of type P3 involves a column-name which is the primary key of the relation.

**Type P4.** The same as type P3 except that the index does not exist. If the P3 predicate is later determined to be "usable," in the sense that its position in the predicate tree allows it to reduce the search space of tuples, it will be converted into a type P3 by dynamic creation of a temporary index. This is done because an index can be created in XRM with a single scan of the relation. Lack of the index would require a scan of the relation for each tuple from the higher level block.

**Type P5.** Predicates which involve a correlation to a higher level block and which cannot be resolved by an index. These are like P2 predicates except that they involve the correlation. P5 predicates require a scan over the relation (or the reduced scan list if available) for every tuple in the higher level block to which the predicate is correlated.

Note that the predicates of nested query blocks are classified during recursive calls to the Optimizer. If such a query block contains no correlation to a higher level block it will be completely evaluated by the call and will be effectively replaced by the evaluation result. If a block involves a higher level correlation, its evaluation must await invocation of the recursive Scanner. The outer (highest level) block cannot contain upward correlations; hence, its predicates can only be of type P1 or P2.

After step A of the Optimizer, we go to step B if there are no correlation predicates (P3, P4, or P5). Otherwise we go to step C.

**Step B**

In step B the query block is known to have only P1 and P2 predicates. Step B will proceed to find the actual tuples which satisfy the query block. This process will, in turn, be done in three steps:

B1. First, we decide which index resolvable predicates (P1's) are capable of limiting the number of tuples which must be fetched to evaluate the block. This process employs the Index Selection Algorithm.

B2. Next, we form a scan list for the query block, consisting of the tid's of those tuples which must be fetched and examined to evaluate the block. This step uses the List Combining Algorithm.

B3. Finally, we actually fetch the tuples on the scan list and test them for satisfaction of the predicate tree by means of the Test Tree Algorithm.

We illustrate this process by means of an example query. Suppose that the query block being processed is Q12 below.

Q12.

```
SELECT *
FROM EMP
WHERE DNO = 19
AND MGR = 525
AND (TITLE= 'DIRECTOR' OR SAL > 20000);
```

Although we have chosen an outer level query block for our example, step B applies equally well to inner,

Fig. 3.

| EMP: | MNO | NAME | DNO | MGR | TITLE | SAL |
|---|---|---|---|---|---|---|
| T1— | 101 | ABEL | 19 | 510 | DIRECTOR | 18000 |
| T2— | 102 | BAKER | 19 | 525 | DIRECTOR | 19000 |
| T3— | 103 | CARLSON | 19 | 525 | CLERK | 8000 |
| T4— | 104 | DOE | 19 | 525 | VICE-PRES | 22000 |
| T5— | 105 | EASTMAN | 20 | 525 | CHAIRMAN | 24000 |

nested query blocks. Suppose that Q12 is to be evaluated with respect to the database of Figure 3, in which the tid's of the tuples are represented as T1, . . . , T5. In step B1, we will decide which indexes are useful in resolving the query block. To do so, we mark each P1 in the predicate tree as resolvable (R) and each P2 as not resolvable (N) and then call the Index Selection Algorithm. This algorithm generates a list, called the P*-list, of predicates which should be resolved by index. The algorithm is applied to the root node of the predicate tree. It may invoke itself recursively for lower nodes. The P*-list is computed according to the following rules:

a. If the node is an R-predicate, then P* = the R-predicate.

b. If the node is an N-predicate, then P* is null.

c. If the node is an AND node, call the algorithm recursively for its immediate left and right descendant nodes, producing predicate lists P*L and P*R respectively. Then P* = the concatenation of lists P*L and P*R. This is because a tuple which fails to satisfy either descendant of an AND node cannot satisfy the AND node and need not be checked against the other descendant. Therefore an index which reduces the scan space for either descendant reduces it for the AND node. The scan list of tuples which satisfy an AND node is the intersection of the lists for the descendants.

d. If the node is an OR node, produce lists P*L and P*R as in c. If either P*L or P*R is null, then P* is null. If neither P*L nor P*R is null, then P* = the concatenation of lists P*L and P*R. This is because a tuple which fails to satisfy one descendant of an OR node may still satisfy the OR if it satisfies the other descendant. A scan list obtained from an index for one descendant of an OR node does not limit the scan space for the other descendant. The scan list of tuples which satisfy an OR node is the union of the lists for the descendants.

In our example Q12, let us suppose that the first three predicates (those on DNO, MGR, and TITLE) are index resolvable (hence we label them R) and the fourth predicate (on SAL) is not resolvable by index (hence we label it N). Then Figure 4 shows how the Index Selection Algorithm computes the P*-list for the query block.

We see from Figure 4 that only the first two predicates have been found useful for limiting the scan list for this query block. The predicate TITLE = 'DIRECTOR', even though it is resolvable by index, cannot limit the set of tuples to be scanned because it is ORed with another non-index-resolvable predicate.

We now proceed to step B2, which will compute the actual scan list for the query block. First, we use the chosen indexes to resolve each predicate on the P*-list into a list of the tid's of tuples which satisfy the predicate. Next, we combine these lists into a scan list for the block by means of the List Combining Algorithm. This algorithm passes over the predicate tree and labels each AND, OR, or predicate node with either R, S, or N. A label of R means that the node and all its descendants are resolved. Associated with an R node is a tid-list of those tuples which satisfy the node. We denote an R node and its associated tid-list Li as R(Li). A label of S means that some descendants of the node are not resolved, but there exists a scan list containing tid's of those tuples which could satisfy the node, excluding those tuples which are known not to satisfy the predicate subtree below the node. An S node and its associated scan list Lj are denoted by S(Lj). A label of N means that the node is not resolved and no scan list smaller than the whole relation is known for the node. The List Combining Algorithm begins by labeling each predicate node either R(Li) or N, depending on whether the predicate has been resolved. The algorithm then labels the AND and OR nodes according to the following rules:

a. If the immediate descendants of the node are not yet labeled, call the List of Combining Algorithm recursively to label them.
b. If the node in question is an AND node, choose its label from the following table:

| | | Label of descendant #2 | | |
|---|---|---|---|---|
| | | R(L2) | S(L2) | N |
| Label of descendant #1 | R(L1) | R(L1 ∩ L2) | S(L1 ∩ L2) | S(L1) |
| | S(L1) | S(L1 ∩ L2) | S(L1 ∩ L2) | S(L1) |
| | N | S(L2) | S(L2) | N |

c. If the node in question is an OR node, choose its label from the following table:

| | | Label of descendant #2 | | |
|---|---|---|---|---|
| | | R(L2) | S(L2) | N |
| Label of descendant #1 | R(L1) | R(L1 ∪ L2) | S(L1 ∪ L2) | N |
| | S(L1) | S(L1 ∪ L2) | S(L1 ∪ L2) | N |
| | N | N | N | N |

Figure 5 shows how the List Combining Algorithm labels the nodes of the predicate tree for our example Q12. The first two predicates are resolved into tid-lists by means of their indexes and the scan list for the query block is found to be the intersection of these two tid-lists.

Fig. 4.

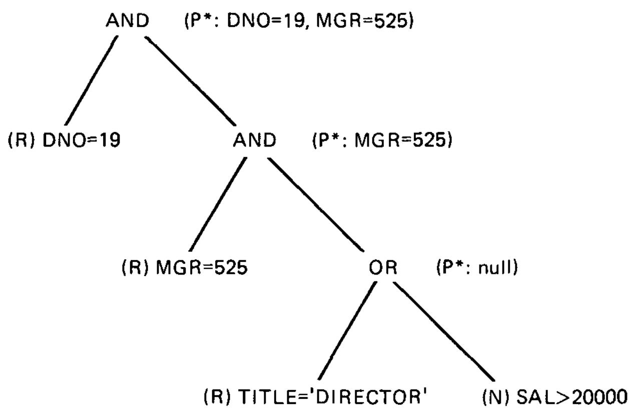
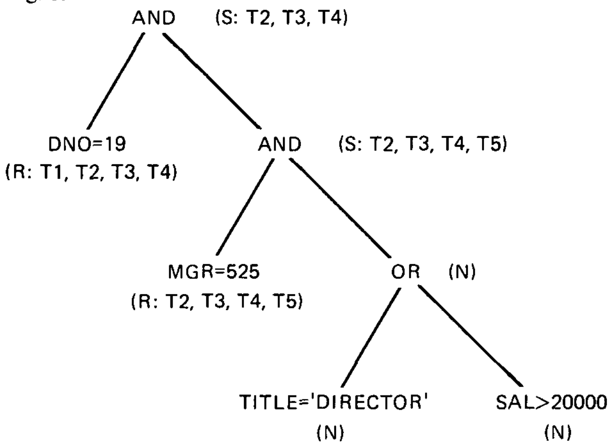


Fig. 5.



The evaluation of the query block is now completed by step B3, which performs the necessary scan. If the query block is already fully resolved into a tid-list (root node is labeled R), then it is only necessary to scan over those tuples on the list and select the indicated fields for output or computation of the indicated built-in function. However, if the root node of the tree is labelled N or S, as in our example, it is necessary to fetch each tuple on the scan list (or the whole relation if the label is N) and test it against the predicate tree by means of the Test Tree Algorithm. The Test Tree Algorithm is a recursive algorithm which is applied to the root node of the predicate tree. It works as follows:

a. If an unresolved node (labeled S or N) has descendants, they are tested by applying the function recursively. The node truth value is then found by applying the node label operator (AND or OR) to the descendant values.

b. For nodes which have an R-list, we merely test the tuple's tid for inclusion in the R-list, returning a true or false value. This is an efficient operation in XRM.

c. For unresolved predicates, we test the actual column values of the tuple against the predicate, returning true or false. If the predicate has a nested query block which has not yet been evaluated, we make the tuple column values available to correlation terms and call the Scanner recursively. This will complete the evaluation of the nested block and we can then complete the testing of the tuple against the predicate. Since testing for inclusion in a list is generally more efficient than testing a tuple against a predicate, the List Combining Algorithm switches descendants of an AND or OR node if necessary when only one descendant is resolved to ensure that the left hand descendant is the resolved one. The Test Tree function checks the left-hand descendant first and can often logically eliminate the need to test the right-hand one.

In our example Q12, step B3 will scan over the tuples T2, T3, and T4 and apply the Test Tree Algorithm to each. Tuple T2 will be successful because it is in the R-lists of the first two predicates and its TITLE is 'DIRECTOR'. Tuple T3 will fail because it fails to satisfy both the predicates on TITLE and SAL. Tuple T4 will succeed. Hence the final result of our query block Q12 is tuples T2 and T4. If the query block had specified some built-in function such as SUM(SAL), it would have been computed incrementally as each tuple was tested against the predicate tree.

### Step C

When the query block has some type P3, P4, or P5 predicates, step C of the Optimizer is called. The current query block is known to be nested inside some higher level block to which it is correlated by one or more predicates. Therefore the current block cannot be completely evaluated at this time. Instead, it will be evaluated repeatedly by the Scanner, once for each tuple in the scan list of the higher level block. We wish to minimize the set of tuples which must be retrieved and tested upon each reevaluation of the current block. This set may be limited in the following ways: (1) by resolving P1's and P2's immediately, since they do not depend upon the higher level block—this resolution will be done once; and (2) upon each reevaluation of the current block, by resolving P3's via their indexes.

Since P1's, P2's, P3's, and P4's (if the P4's are converted to P3's) are potentially capable of reducing the size of our scan list upon each reevaluation, we wish to find out exactly which ones do in fact limit the scan list by virtue of their position on the predicate tree. This may be done by marking all P1's, P2's, P3's, and P4's as resolvable (R) and all P5's as not resolvable (N) and calling the Index Selection Algorithm. The resulting P* list tells us which predicates to resolve in advance of reevaluation or via index during reevaluation. All P1's on the P*-list are resolved immediately. If any P1's were resolved, we call the List Combining Algorithm to generate a scan list. If there are any P2's, a special single scan of the scan list is called and the tuples are tested against each P2, generating a resolved tid-list for each such P2. If a P2 contains a nested query block with correlations to the current block, the Scanner is called recursively, as in step C of the Test Tree function. The P2 lists are then combined with P1 lists, if possible, by the List Combining Algorithm. If there are P4's on the P* list, they are converted to P3's by creation of a temporary index. This is as far as we can go in preparing the current block for repeated evaluation by the Scanner. If the current block is nested in a higher level block, control is returned to the Optimizer predicate classification function at the higher level with a notation that reevaluation is necessary.

When the Scanner is called for the outer (highest level) block at the end of Optimizer step B, it initiates a scan of the tuples on the scan list if the block is not yet fully resolved. This function has been described in the description of step B. When the Scanner is called for the reevaluation of a nested block in the context of a higher level scan, several preparatory steps should take place:

a. Resolve all P3's via their index. This is possible because a tuple from the higher level block is available.
b. If there are P3's, then call the List Combining Algorithm to generate a new, more restrictive scan list.
c. If the block is not yet resolved (because of P5's), initiate a scan of the scan list. This may involve calling the Scanner recursively for nested queries containing correlations.

Note again that the Scanner performs the function of generating the requested form of output from each query block. This may be a built-in function or a temporary relation containing some or all of the columns of the designated relation. If a scan of the scan list is required, the output is generated as tuples are qualified. If no scan is required, a special output generation scan may be initiated.

## Future Plans

Tests with a thousand-tuple database indicate that use of indexes does speed up retrieval, even when all the pages containing the relation have to be accessed to finally retrieve the tuples which qualify.

We are now engaged in the design of a multiple-user SEQUEL research prototype. We hope that this system will demonstrate the practical feasibility of a relational system for both application programs written by programmers and ad hoc queries written by non-programmers. It will have the interlocks required for shared access by several users. It will also have the authorization and recovery features required by a multiple-user environment. We are planning to provied facilities for making integrity assertions about the database which are automatically checked on update.

A more powerful and efficient relational memory system will evolve from the XRM experience.

Improvements and additions to the implemented version of the SEQUEL language will be made to bring it closer to the published version [5]. In particular, arithmetic operators and the GROUP BY operator will be implemented. Another important feature will be efficient coupling to a host language, such as APL or PL/I, so that the full power of a programming language can be applied to the data returned by a SEQUEL query.

*Acknowledgments*

The authors wish to acknowledge the following contributions in the preparation of this paper. Ray Boyce provided his managerial skills and wrote a number of the storage accessing subroutines until his untimely death in June 1974. Paul Fehder participated in the system design and wrote the parser semantics and the data definition and maintenance routines. Raymond Lorie maintained the XRM interface and provided very helpful advice during debugging. W.F. King, our department manager, contributed significantly to the optimization algorithms as well as providing managerial support.

**References**

**1.** Bjorner, D., Codd, E.F., Deckert, K.L., and Traiger, I.L. The Gamma-0 N-ary relational data base interface: specifications of objects and operations. Res. Rep. RJ 1200, IBM Research Laboratory, San Jose, Calif., April 1973.
**2.** Boyce, R.F., Chamberlin, D.D., King, W.F., and Hammer, M.M. Specifying queries as relational expressions. Proc. ACM SIGPLAN/SIGIR Interface Meeting, Gaithersburg, Md., Nov. 1973.
**3.** Boyce, R.F. and Chamberlin, D.D. Using a structured English query language as a data definition facility. Res. Rep. RJ 1318, IBM Research Laboratory, San Jose, Calif., Dec. 1973.
**4.** Bracchi, G., Fedeli, A., and Paolini, P. A language for a relational data base management system. Proc. Sixth Annual Princeton Conference on Information Science and Systems, March 1972, pp. 84–92.
**5.** Chamberlin, D.D. and Boyce, R.F. SEQUEL: A structured English query la nguage. Proc. 1974 ACM SIGFIDET Workshop, Ann Arbor, Michigan, April 1974, pp. 249–264.
**6.** Codasyl Data Base Task Group Report. ACM, April 1971.
**7.** Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
**8.** Codd, E.F. Relational completeness of data base sublanguages. *Courant Computer Science Symposia, Vol. 6: Data Base Systems.* Prentice-Hall, Engelwood Cliffs, N.J. 1971.
**9.** Codd, E. F. A data base sublanguage founded on the relational calculus. Proc. 1971 ACM SIGFIDET Workshop, San Diego, Calif, Nov. 1971, pp. 35–68.
**10.** Goldstein, R.C., and Strnad, A.L. The MACAIMS data management system. Proc. 1970 ACM SIGFIDET Workshop, Houston, Texas, pp. 201–229.
**11.** Lalonde, W.R., Lee, E.S., and Horning, J.J. An LALR(*k*) parser generator. Proc. IFIPS Congress, 1971. Ljubljana, Yugoslavia. North-Holland Publishing Co., Amsterdam, 1972, 513–518.
**12.** Lorie, R.A. XRM—An extended (N-ary) relational memory. Tech. Rep. 320–2096, IBM Scientific Center, Cambridge, Mass., Jan. 1974.
**13.** Lorie, R.W., and Symonds, A.J. A relational access method for interactive applications. In [8].
**14.** Peteul, B., and Lorie, R.A. Multisegment relational memory users guide, IBM internal rep., Oct. 1973.

Programming Techniques

G. Manacher Editor

# Merging with Parallel Processors

Fănică Gavril
University of Illinois

**Consider two linearly ordered sets $A$, $B$, $|A| = m$, $|B| = n$, $m \leq n$, and $p$, $p \leq m$, parallel processors working synchronously. The paper presents an algorithm for merging $A$ and $B$ with the $p$ parallel processors, which requires at most $2\lceil \log_2(2m+1) \rceil + \lfloor 3m/p \rfloor + \lceil m/p \rceil \lceil \log_2(n/m) \rceil$ steps. If $n = 2^\beta m$ ($\beta$ an integer), the algorithm requires at most $2\lceil \log_2(m+1) \rceil + \lceil m/p \rceil (2+\beta)$ steps. In the case where $m$ and $n$ are of the same order of magnitude, i.e. $n = km$ with $k$ being a constant, the algorithm requires $2\lceil \log_2(m+1) \rceil + \lceil m/p \rceil (3+k)$ steps. These performances compare very favorably with the previous best parallel merging algorithm, Batcher's algorithm, which requires $n/p + ((m+n)/2p)\log_2 m$ steps in the general case and $km/p + ((k+1)/2)(m/p)\log_2 m$ in the special case where $n = km$.**

**Key Words and Phrases$_n$ parallel processing, parallel merging, parallel binary insertion**

**CR Categories: 5.31**

This work was supported in part by the National Science Foundation under Grant No. GJ-41538.

Part of this work was done while the author was at the Weizmann Institute of Science, Rehovot, Israel. Author's present address: Department of Mathematical Sciences, Tel-Aviv University, Ramat-Aviv, Tel-Aviv, Israel.