1 INTRODUCTION

The relational algebra invented by Codd [10,11,12] is fundamental to the theory and implementation of relational data bases [7]. Relational algebra is extensively used for investigations into the structure of relational data bases and the design of relations, particularily with respect to relational decomposition to eliminate undesirable functional [1,2,3,10], multivalued [13], and join and generalized dependencies [17,18]. It is also extensively used as the intermediate target language [20] in the reduction of expressions of nonprocedural langauges used with relational data bases, such as SQL [8, 14], originally developed at IBM, and QUEL [19], the non procedural language of INGRES.

Relational algebra is ideal as the target language for reduction of SQL expressions, and common SQL expressions can readily be reduced to quite simple algebra expressions [14,21]. SQL is easily the most important non procedural relational language, and has been implemented by many vendors. A standard version is being developed by ANSI. However, SQL has its roots in conventional predicate calculus [11] which permits only the existential and universal quantifiers. These basic quantifiers are necessary and sufficient in predicate calculus, and although SQL does not use quantifiers as such, it has structures that correspond to existentially and universally quantified sets of tuples [14,20,22] Thus SQL is essentially a primitive non procedural language. This primitive nature of SQL is advantageous when it comes to reduction of common SQL expressions [14,23], but has the disadvantage of frequently requiring highly contrived SQL expressions, which are difficult to reduce to relational algebra, for certain types of what are otherwise quite simple requests. Such requests involve any of the large number of natural quantifiers [14, 16] available to the user of natural language.

An example will illustrate this important point. For example, if we take the database DEPT[DEPT#, TYPE], EMP[EMP#, DEPT#, SALARY], the simple request:

> "Find each marketing (TYPE) department, in which most employees earn more than $20,000"

requires the SQL expression:

```
SELECT DEPT# FROM DEPT
WHERE TYPE = 'MARKETING'
AND  (SELECT COUNT (*) FROM EMP
      WHERE SALARY  > 20,000
      AND DEPT.DEPT# = EMP.EMP#)
      >
      (SELECT COUNT (*) FROM DEPT
       WHERE SALARY <= 20,000
       AND DEPT.DEPT# =  EMP.EMP#)
```

which few, if any, current SQL implementations can reduce. The problem in this example is the implicit natural quantifier "for most", which is not allowed in SQL. The SQL user must instead specify that the count of associated employees earning more than $20,000 exceeds the count of those not earning more than $20,000. Thus, although the SQL is correct, it is also contrived.

In order to eliminate this difficulty, the author has researched and developed a class of natural quantifier non procedural data base languages, which permit the application of natural quantifiers to associations between relations in a natural manner - at least in the author's opinion. One of these languages, called SQL/N [4,5], has served as a working tool for research and development, and has the advantage of being upward compatible with SQL. SQL/N is not the subject of this paper, although it explains the motivation; we also need to use SQL/N, in a minimal context, to illustrate the use of our proposed extension to relational algebra.

Having defined SQL/N, the next problem is also one of software engineering, namely implementation techniques.

There are two ways to implement a natural quantifier (NQ) language that is upward compatible with SQL. One way is to treat the NQ language as a front end, and develop a reduction system for converting NQ expressions to SQL expressions.

The problem with this is that the SQL expression generated, when natural quantifiers are used in the NQ expression, is likely to be complex, along the lines of the SQL expression above, and will in turn require a further reduction to relational algebra that will be either time-consuming, or not possible with existing SQL implementations. In other words, this approach is likely to yield either a very slow reduction system, or not work. In addition, it is not interesting.

The other approach is to develop a reduction system capable of reducing NQ expressions directly to relational algebra expressions. It is work with this second method that has led us to propose an extension to relational algebra that greatly simplifies the reduction process.

2 A GROUP SELECTION COMMAND FOR RELATIONAL ALGEBRA

We have stated that simple natural quantifier request often require complex SQL expressions that current SQL implementations cannot reduce. Unfortunately, it is also the case that simple natural quantifer requests often require quite complex relational algebra expressions and procedures, and sometimes such algebra procedures cannot be found at all. As an example, consider the request and database of the previous section. We use the abreviations D[D#, T], E[E#, D#, S] for the data base.

To construct the algebra procedure, the first step would be to eliminate from E, those employees in departments that are not in marketing (abreviated to T = 1). The required version of E, or E_1, is given by:

$$E_1 = \P_{E\#,D\#,S}[\ (\S_{T\ =\ 1}(D))\ *\ E]$$

Here § denotes the algebraic selection operation [10], and ¶ the projection operation [10].

The next step is to extract from E_1, the relation E_m for employees that earn more than $20,000 and the relation E_n for employees that do not. These relations are given by:

$$E_m = \P_{E\#,D\#} [\S_{S > 20,000} (R_1)]$$

$$E_n = \P_{E\#,D\#} [R_1] - E_m$$

where we have eliminated the S attribute from futher consideration, as it is no longer useful. Instances of E_m and E_n could be:

E#	D#
e1	d1
e4	d1
e6	d2

E_m

E#	D#
e3	d1
e5	d2
e7	d2

E_n

Of the two departments shown, d1 has two employees making more than $20,000 and only one that is not, the converse being true for d2. For these instances, d1 is the answer the request.

Unfortunately, there is no relational algebraic operation that can be carried out on relations E_m and E_n that will give each D# value such that the number of tuples in E_m with such a D# value exceeds the number in E_n. In contrast, there is no problem with devising algorithms for this purpose, when the two relations E_m and E_n are processed as sequential files. The problem with the relational algebra as it is conventionally constituted is that we have no method of specifying a quantity of tuples within a relation. It is precisely a quantity of tuples within a relation that is specified in an SQL/N expression containing a natural quantifier. For example, <u>for most</u> means "more than half", and so on.

An extension to relational algebra that permitted quantities of tuples to be specified would clearly be useful. The question is whether the extension should be a low level one, involving a COUNT function, along the lines of that used in SQL [8, 14], a higher level one involving the natural quantifiers. Because our motivation is the efficient reduction of NQ expressions, such as SQL/N expressions, we favor the NQ approach to an extension. Our extension proposal merely involves introduction of a single new command, called a group-select

operation.

The semantics of the parameters of the group-select operation are:

Operation: Group-select (q, c, r, a)

q : specification of number of tuples satisfying a condition, that is, any quantifier.

c: the logical condition to be satisfied by q tuples.

r: the relation from which groups of tuples are being extracted.

a: the attribute in r whose value defines a group of tuples being tested for extraction.

In other words, the operation partitions a relation according to the values of a given attribute a, and extracts each entire block of tuples (from the partition) for which a specified number of tuples (q) satisfy a condition (c).

We can immediately apply this command to the relation E_1, in order to complete the retrieval left unfinished above. The answer to the request is the set of D# attribute values in the relation obtained by:

group-select(for most, (S > 20,000), E_1, D#)

The semantics are:

select each group of tuples from E_1 with the same D# value, provided that for most tuples of the group, S > 20,000.

Thus the quantifier is for most, the condition that the specified quantity of tuples must meet is (S > 20,000), the relation is E_1 and the attribute used for grouping the tuples is D#. These four parameters are necessary and sufficient.

As far as a suitable algebra syntax is concerned, we suggest the syntax shown below in the complete algebraic procedure for carrying out the request from the first section. The group-select operation has the symbol Γ.

$$R_1 = \Pi_{E\#,D\#,S}[\ (\sigma_{B = 1}(D)) * E]$$

$$R_2 = \Gamma_{\exists(>\forall/2)[S > 20,000]}(R_1)_{D\#}$$

$$R_3 = \Pi_{D\#}(R_2)$$

A complete and consistent notation for the natural quantifiers has also been developed. The universal quantifier for all is denoted by either the conventional ∀ or our ∃(∀). For all but 1 becomes ∃(∀ - 1), and for most ∃(>∀/2). The existential quantifer can be denoted by either the conventional ∃ or our ∃(∃) or even ∃(>1). A list of the symbols for the more common natural quantifers is given in the appendix to this paper.

3 REDUCTION OF SQL/N EXPRESSIONS

It is not our purpose to report on a complete reduction algorithm for SQL/N in this paper. SQL/N permits the application of natural quantifiers to every kind of association [5,6,9] that can occur in a relational database, whether cyclic or non cyclic, primitive or non primitive, composite or non composite [6], and each type of assocaition requires unique reduction techniques. But to demonstrate the utility of this modest extension of relational algebra, we can give the reduction expression for a general SQL/N expression involving the simplest and most common association between the tuples of two relations, namely the common or primitive one-to-many association. This is the association between DEPT and EMP in the example from the fist section.

Assume relations **A**(A,C), **B**(B, G, F), with primary key attributes underlined, and involved in a one-to-many association, such that for one **A** tuple there may be zero or more associated **B** tuples, and let the rule for primitive association be that the **A** candidate key attribute A be drawn on the same domain as the **B** attribute G, and that A and G attribute values be equal. We call this rule W, and assume it specified in the relational schema. (W can also be called a coincidental mode of association [5].)

The general SQL/N expression for retrieval of **A** tuples depending on **A** attribute values and associated **B** tuples is:

SELECT * FROM **A**

WHERE A-condition AND quantifier W-ASSOCIATED **B** (B-condition)

Here A-condition is a logical expression, compound or atomic, involving the

attributes of **A**, and B-condition is a logical expression for the attributes of **B**. Readers may be surprised at the simplicity of this general expression, but should not be misled. The expression is also powerful. The simplicity arises from the fact that retrievals of the same semantic class in English always give rise to SQL/N expression of the same syntactic class [5], which is not the case with SQL. For example, taking the data base and request from the first section, the SQL/N expression gives us an instance of the general expression above:

```
SELECT *  FROM DEPT
WHERE TYPE = 1 AND FOR MOST W-ASSOCIATED EMP (SALARY  > 20,000)
```

We had the corresponding SQL expression in the first section. As a second example, suppose that we have the request:

" Retrieve the DEPT records for departments in engineering in which all but two employees earn more than $30,000".

The SQL expression is another instance of the general expression above:

```
SELECT * FROM DEPT
WHERE TYPE = 3 AND FOR ALL BUT 2 W-ASSOCIATED EMP (SALARY  > 30,000)
```

or equivalently:

```
SELECT * FROM DEPT
WHERE TYPE = 3 AND FOR EXACTLY 2 W-ASSOCIATED EMP (SALARY  < = 30,000)
```

Here the SQL expression has a different structure from the previous SQL expression:

```
SELECT * FROM DEPT
WHERE TYPE = 3  AND 2 =   (SELECT COUNT(*) FROM EMP
                             WHERE SALARY  < =  30,000
                               AND DEPT.DEPT# = EMP.DEPT#)
```

Many current SQL systems would probably not be able to reduce this expression either.

The general SQL/N expression given above always reduces to the following relational algebra procedure containing a group-select operation:

$$R_1 = (\S_{\text{A-condition}}(\mathbf{A})) * \mathbf{B}$$

$$R_2 = \P_{A,C}[\Gamma_{(\text{quantifier})[\text{B-condition}]}(R_1)_G]$$

In the case of the second retrieval above, the SQL/N expression would thus reduce to:

$$R_1 = (\S_{T=3}(D)) * E$$

$$R_2 = \P_{D\#,T}[\Gamma_{\exists(\forall\ -\ 2)[S\ >\ 30{,}000]}(E)_{D\#}]$$

Readers should evaluate it and determine that it is correct, and for comparison purposes attempt a reduction of the corresponding SQL expression - without the group-select operation.

In the setting up these algebra expressions the reduction system would inspect the W specification to determine the join attributes for the join in the first expression. In the general case these are the attributes A and G. The reduction system would also inspect W to determine the tuple grouping attribute G for the group select operation in the second expression. The specification in the schema for W simply states that an **A** and **B** tuple are W-associated iff they have equal A and G values (a common domain being assumed). Assuming r_A, r_B to be **A** and **B** tuples, a typical W specification in the database schema would be equivalent to:

$$\forall r_A \forall r_B[(r_A.A = r_B.G) \quad <=> \quad W(r_A.A,\ r_B.B)]$$

for the primitive one-to-many association we are assuming. For other types of association the schema W specification would be different [5].

4 PERFORMING THE GROUP-SELECT OPERATION

It can be seen that, for the primitive one-to-many association, the reduction of a very large and important class of SQL/N expressions becomes trivially simple when a group-select operation is available in relational algebra. An obvious question is the efficiency with which a group-select operation can be carried out.

Unfortunately, the answer is that the operation cannot be carried out efficiently with conventional hardware. However, file processing algorithms for carrying out the operation are trivially easy to construct. They are just all expensive to execute. Without secondary indexes, the basic method of carrying out the group-select operation essentially involves sequential processing of the file for the relation, counting records that meet conditions, and the number of records in a group. This is clearly expensive, especially when the file is large.

Note, however, that the expense of carrying out the group-select operation is not something inherent in the natural quantifier approach. Suppose that we do not use an SQL/-like language, but use SQL instead. If natural quantifiers are involved in the retrieval request, the COUNT function has to be used in some way in the SQL expression, thus also requiring that tuples be counted, often by sequential file processing. The problem is inherent in retrievals involving natural quantifiers, and not in the language used to express them. This ultimate need for counting of tuples, and resulting inefficiency, may well be the reason for the current commercial inertia with respect to natural quantifier facilities in data base langauges. Natural quantifer languages, like SQL/N, make natural quantifier requests trivially easy to formulate, but result in retrieval routines that are inherently costly, the major cost involving the performance of the group-select operation, or its equivalent.

The method of secondary indexes can be employed to much improve the efficiency of the group-select operation only where updates are limited. The method involves creating a secondary index on the tuple grouping attribute G, as well as on any attributes involved in the quantifier conditions, such as the F attribute in the examples. (F corresponds to SALARY). For any given G value, the primary keys of the group of **B** tuples can be obtained from the G secondary index by direct access. Using direct access with the F value in the B-condition (for example $F = k_2$) to the F secondary index, we obtain the **B**

primary keys for tuples where F = k_2. Intersecting these keys with those ontained from the G secondary index gives the primary keys of the tuples in the group with the given G value that meet the quantifier condition. From this the system can easily determine if the quantity of tuples is sufficient - for example, if there are at least 3, or if there are a majority, that meet the condition B-condition.

Any method of improving efficiency based on secondary keys is unsatisfactory in general because of the updating cost, so that a better method must be found. The only other alternative is associative memory with a data base machine, and research in this direction is currently under way.

5 CONCLUDING REMARKS

An efficient method of reducing natural quantifer language expressions has been found, and requires only that relational algebra be extended to permit a group select operation, which extracts from a relation each group of tuples in which a specified quantity of tuples, specified using a natural quantifier, satisfies specified conditions. Without this operation, it is very difficult to reduce NQ expressions to relational algebra expressions, often for the reason that no relational alebra expression is possible. The general SQL/N expression given in this paper, and reduced to a general algebra procedure, is for a particular class of natural quantifier expressions, although probably the most common class. Because of the wide scope of SQL/N and the fact that it permits application of natural quantifiers to any of a large number of association types [6], there are other types of reductions. With all of the associations currently known, reduction to relational algebra expressions containing group-select operations has proven quite straightforward.

A remaining problem is an efficient method of carrying out group select operation. It seems likely that the only satisfactory solution will involve associative memory and data base machines. In the meantime, it is a simple matter to write a sequential file processing program that will carry out a group-select operation with any of the large number of natural quantifiers. The problem is that such a program will be inherently expensive to execute.

Appendix 1. Common natural quantifiers

1. FOR n, FOR THE n, FOR EXACTLY n	$\exists(n)$
2. FOR AT LEAST n, FOR n OR MORE	$\exists(\geq n)$
3. FOR AT MOST n, FOR n OR LESS	$\exists(\leq n)$
4. FOR AT LEAST 1, FOR ONE OR MORE, FOR SOME	$\exists(\exists), \exists, \exists(\geq 1)$
5. FOR BETWEEN n AND m	$\exists(>n \wedge <m)$
6. FOR ALL, FOR EACH, FOR ALL IF ANY, FOR EACH IF ANY	$\exists(\forall), \forall$
7. FOR ALL BUT n	$\exists(\forall - n)$
8. FOR ONE AND ALL	$\exists(\geq 1 \wedge \forall), \exists(\exists \wedge \forall)$
9. FOR NO	$\exists(0)$
10. FOR SOME BUT NOT ALL	$\exists(\geq 1 \wedge \neg \forall), \exists(\exists \wedge \neg \forall)$
11. FOR SOME BUT NOT n	$\exists(\geq 1 \wedge \neg n), \exists(\exists \wedge \neg n)$
12. FOR SOME BUT NOT MORE THAN n	$\exists(\geq 1 \wedge \leq n), \exists(\exists \wedge \leq n)$
13. FOR SOME BUT LESS THAN n	$\exists(\geq 1 \wedge <n), \exists(\exists \wedge <n)$
14. FOR MOST, FOR A MAJORITY OF	$\exists(>\forall/2)$
15. FOR A MINORITY OF	$\exists(<\forall/2)$
16. FOR x PERCENT OF, FOR EXACTLY x PERCENT OF	$\exists(x/100)$
17. FOR AT MOST x PERCENT OF FOR x PERCENT OR LESS OF	$\exists(\leq x/100)$
18. FOR AT LEAST x PERCENT OF FOR x PERCENT OR MORE OF	$\exists(\geq x/100)$
19. FOR BETWEEN x AND y PERCENT OF	$\exists(>x/100 \wedge <y/100)$

REFERENCES

1. Aho, A. V., Beeri, C., and Ullman, J.D. The theory of joins in relational databases, ACM Trans. Database Syst., 4(3), 1979, 317-314.

2. Armstrong, W.W. Dependency structures of database relationships, Proc. IFIP 74, North Holland, Amsterdam, 1974, 580-583.

3. Bernstein, C.W., and Chiu, C.W. Using semijoins to solve relational queries, J. ACM, 28(1), 1981, 25-40.

4. Bradley, J. SQL/N and attribute/relation associations implicit in functional dependencies, Int. J. Computer & Information Science, 12(2), 1983

5. Bradley, J. SQL/N and modes of association in relational databases, Research Report No. 84/143/5, Univ. of Calgary, Calgary, Alberta, Canada, 1984, 39 pages.

6. Bradley, J. A fundamental classification of assocaitions in relational databases, Research Report No. 85/204/17, Univ. of Calgary, Alberta, Canada, 1985, 32 pages.

7. Chamberlin, D.D. Relational database management systems. Comput. Surv., 8(1), 1976, 43-66.

8. Chamberlin, D.D., et al. SEQUEL 2: A unified approach to data definition, manipulation and control, IBM J. Res. & Dev., 20(6), 1976, 560-575.

9. Chen, P.P., The entity-relationship model: Towards a unified view of data, ACM Trans. Database Syst. 1(1), 1976, 9-36.

10. Codd, E.F. Further normalization of the database relational model, In "Database Systems", Courant Computer Science Symposium, 6, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1971, 33-74.

11. Codd, E.F. Relational completeness of database sub-languages, In "Database Systems", Computer Science Symposium 6, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1971, 65-98.

12. Codd, E.F. Relational database: A practical design for productivity, CACM, 25(2), 1982, 109-117.

13. Fagin, R. Multivalued dependencies and a new normal form for relational databases, ACM Trans. Database Syst., 2(3), 1977, 262-278.

14. Kim, W. On optimizing an SQL-like nested query, ACM Trans. Database Syst.,

7(3), 1982, 443-469.

16. Merret, T.H. QT logic: Simpler and more expressive than predicate calculus, Information processing letters, 7(6), 1978, 251-255.

17. Sadri, F, Ullman, J. D.. Templete dependencies: a large class of dependencies in relational data bases and its complete axiomatization. J. ACM 29(2), 363-372, 1972.

18. Sagiv, Y. and Walecka, S.F. Subset dependencies and a completeness result for a subclass of embedded multivalued dependencies, J. ACM 29(1), 1982, 363-372.

19. Stonebraker, M., Wong, E., Kreps, P., and Held, G. The design and implementation of INGRES, ACM Trans. Database Syst., 1(3), 1976, 189-222.

20. Ullman, J.D., Principles of Database Systems, Computer Science Press, Rockville, MD, 1983.

21. Wald, J.A., and Sorenson, P.G. Resolving the query inference problem using Steiner trees, ACM Trans. Database Syst. 9(3), 348-368, 1984.

22. Weiderhold, G. Database Design, McGraw-Hill, New York, 1983.

23. Welty, D. and Stemple, D.W. Human factors comparison of procedural and non procedural query languages, ACM Trans. Database Syst., 6(4), 1981, 626-649.