数理解析研究所講究録
第 666 巻 1988 年 245-254

Efficient Processing of Set Oriented SQL Queries

Mohamed El-Sharkawi Yahiko Kambayashi
Dept. of Computer Sci. and Communication Eng.
Kyushu University, Fukuoka , Japan

Abstract

In this paper, we discuss efficient processing of set queries. Our approach is to convert, if possible, a set query into a non-set query. To do the conversion we use set semantics, precisely set sizes. For instance, for two sets to be equal, they must have same size. In the relational model, the size of set of values of some attribute that is associated with a value of another attribute can be reasoned using functional dependencies. Most of set queries are convertible except one case when the two sets are of size greater than one. However, we can do two preprocessing steps to speed up processing the query and detect null answers as early as possible.

1- Introduction

One of the advantages of the relational model of databases [5] is that it supports nonprocedural query languages to manipulate stored data. A component of the DBMS, the query optimizer, is responsible to process a given query in an efficient way. There are several techniques to optimize a query [8]. Another advantage of the relational model is the capability of manipulating sets by supporting set operations, i.e. set containment and set comparison operations. While efficient processing of queries is discussed in deep, efficient processing of set queries does not receive a lot of attention. Set queries are submitted frequently in real applications of databases and usually the manipulated sets are of large sizes. In some extension of the relational model, namely the N^2F^2 model in which attributes are not restricted to be atomic values, may be relations, lists, or sets, set queries are necessary to access such data. Last, to answer universally quantified queries in knowledge-base systems, the query is later mapped to a set query. Thus processing such queries efficiently is critical for system performance improvement. Query languages for the relational model should support set operations. SQL [4] (Structured Query Language), a query language implemented at IBM [1]. Now, it is considered as a standard query language. SQL supports set operations and a query which is a set query should be written in the nested form. That is the query is written as two query blocks connected by the set operation. There are two possible set operations, set comparison and set containment. The general form of set queries in SQL are as follows:

(1) Set comparison queries:

Q.B.1 Comparison Operator ALL Q.B.2.

This query means that, the set of values is the WHERE clause of Q.B.1 (called Outer Query Block OQB) is related to the set of values resulting from Q.B.2 (called Inner Query Block IQB) with comparison operator. The set of comparison operators are $=, \neq, >, <, >=, <=$. Throughout the paper, we concentrate on queries with comparison operators $=$ and $>$.

(2) Set containment queries:

Q.B.1 IS IN / IS NOT IS Q.B.2.

This query means that, the set of values is the WHERE clause of Q.B.1 IS IN/IS NOT IN the set of values resulting from Q.B.2.

In this paper, we give procedures that convert a set query into another nested or unnested query without a set operation. We exploit the query semantics and functional dependencies between attributes to do such conversion. Examples in the paper are based on the familiar Supplier-Part database [6]. The paper is organized as follows. Section 2 gives basic background to this work. Section 3 presents previous work. Section 4 gives the basic idea to convert a set query into a non-set query and also give classification of set queries. Section 5 gives procedures used to process the first type of set comparison queries, that is correlated set comparison queries. Section 6 gives procedures used to process the second type of set comparison queries, that is uncorrelated set comparison queries. In Section 7 procedures to handle set containment queries are given. Section 8 is the conclusion.

2- Basic Concepts

2-1 Relational Schemas and Functional Dependencies (FD)

A *relation schema* $\underline{R}$ is a finite set of attributes A_i, $i=1,\ldots,n$, and denoted by $\underline{R}(A_1,A_2,\ldots,A_n)$. Associated with each attribute A_i a domain D_i from which the attribute obtains its values. A relation R over a relation scheme $\underline{R}$ is a set of finite tuples. A tuple is a mapping that assigns for each attribute A_i a certain value from the associated domain D_i. Union of two sets of attributes X and Y, $X \cup Y$, is the concatenation of attributes in X and Y, and denoted XY. A *relational database scheme* is a collection of relation schemas.

A *functional dependency* (fd) between two sets of attributes X,Y is denoted $X \rightarrow Y$, it means that for any two tuples t_i and t_j, if $t_i[X]=t_j[X]$ then $t_i[Y]=t_j[Y]$; where $t[X]$ means restricting attributes of the tuple t to be X.

For a relation scheme $\underline{R}(X)$, where X is the attributes set of $\underline{R}$, and attributes K, K is a subset of X, we say that K is a *key* of $\underline{R}$, iff $K \rightarrow X$ and there is no proper subset K' of K such that $K' \rightarrow X$.

2-2 Group-by Operation

In the procedures given to convert a set query into another one without a set operation, we apply an operation called group-by [6] on one or both relations in the query blocks. The group-by operation rearranges the relation into groups, such that within any one group all rows have the same value for a specified attribute called the group-by field. The new relation is called the unnormalized form of the original one.

2-3 Syntax of SQL

In this section we describe the syntax of SQL that is relevant to our discussions. In SQL the basic query consists of three clauses: SELECT, FROM, and WHERE. These three clauses constitute a query block. The SELECT clause enumerates output attributes. FROM clause contains names of relations involved in the query. The WHERE clause contains condition that must be satisfied by the query. A condition may contain one of the comparison operators $=, \neq, <, \leqq, >, \geqq$. If there is more than one condition to be satisfied, they may be combined by logical operations AND, OR, and NOT. Beside those comparison operators, there are also set operators. They have the form, IS IN, IS NOT IN, =ALL, $\neq$ALL, =ANY, and $\neq$ANY. In the WHERE clause, a predicate, relevant to this work, to be satisfied may be one of the following:

(a) A simple predicate: Attribute <Comparison operator> Constant value.

(b) A join predicate: Attribute 1 <Comparison operator> Attribute 2.

(c) A nested predicate: Attribute / Constant <Comparison operator> Query block.

(d) A set predicate: Attribute / Constant <Comparison operator ALL / Comparison operator ANY / IS IN / IS NOT IN> Query block / A set of constant.

3- Previous Work

As shown in previous sections, a set query in SQL is written in the nested form. Processing nested queries was discussed in [10]. Where nested queries are classified into five types. Four correspond to whether the inner query block refers to a relation in the outer block and whether the SELECT clause of the inner block contains an aggregate function. A query is of type-N, if the inner block does not refer to the outer block and its SELECT clause does not contain an aggregate function. A query is of type-J, when its inner block refers to the outer block and its SELECT clause does not contain an aggregate function. A query is of type-A, when it is of type-N with aggregate in the SELECT clause of the inner block, and is of type-JA, when it is of type-J with aggregate in the SELECT clause of the inner block. Processing a nested query is based on converting the query into its equivalent unnested one that can be processed efficiently using the underlying query optimizer. The fifth type of nested queries is type-D queries, it has the following form:

```
SELECT RI.CK
FROM RI
WHERE  (SELECT RJ.CH
           FROM  RJ
           WHERE  RJ.CN = RI.CP)
                   OP
           (SELECT   RK.CM  FROM RK),
```

where OP may be a scalar comparison operator, set comparison operator (=, ˆ=), or set membership operator (IS IN, IS NOT IN). It is processed by converting the query into type-N query in the form:

```
SELECT RI.CK
FROM RI
WHERE RI.CP =  (SELECT C1      FROM  RT);
```

where RT(C1) is obtained as

```
(SELECT RJ.CN
FROM RJ RX
 WHERE     (SELECT RJ.CH
            FROM RJ RY
            WHERE RY.CN = RX.CN)      OP
                (SELECT RK.CM   FROM RK).
```

Relation RT is constructed by evaluating a query, that has a form same as the original one. It is not discussed how this query is processed.

In our approach to process set queries, we use functional dependencies to transform, in most of the cases, a set query into a nested query without a set operation. Using functional dependencies in query processing is discussed in [9], to covert a cyclic query [2] into a tree query that is processed efficiently [7].

4- Basic Idea and Classification of Set Queries

4-1 Basic Idea

The basic idea of this work is to convert a set query into a non-set query that can be processed efficiently. To do the conversion, semantics of the query is used. Precisely, set sizes play the main role to decide if some operation is applicable or not. For instance, to check two sets for equality, at first the both should be of same size, otherwise we can reason that they can never be equal. To determine these semantics in a query, function dependencies between attributes in the query are used. A function dependency between attributes U and K ($U \rightarrow K$) says that the set of K-values associated with any u value is for sure of size one. On the other hand, if such dependency does not exist, it means that the set of K-values associated with a u value may be of size greater than or equal to one. By using these semantics we can convert most of the set queries into non-set queries.

4-2 Classification of Set Queries

We classify set queries, type-S , into two subtypes set comparison (type-SCOM) and set containment (type-SCON) queries. In each subtype we distinguish between two situations depending on whether the inner block refers to a relation in the outer block, or does not. A query in which the inner block refers to the outer block is called a correlated query, otherwise is called uncorrelated query. A correlated set comparison query is called type-SCOMCQ, an uncorrelated set comparison query is called type-SCOMUCQ. A correlated set containment query is called type-SCONCQ, an uncorrelated set containment query is called type-SCONUCQ. When the WHERE clause of the outer block of a correlated query contains a "Constant" the query is called of type-SCOMCQ1 in case of set comparison queries, and type-SCONCQ1 in case of set containment queries; when this WHERE clause contains an attribute RI.K the query is called of type-SCOMCQ2 in case of set comparison queries, and type-SCONCQ2 in case of set containment queries. In case of uncorrelated queries when the WHERE clause of the outer block contains a constant value instead of an attribute RI.K, the result is always either true or false thus such a query has no meaning. We distinguish between two cases. When the inner block does not contain a WHERE clause, we call the query of type-SCOMUCQ1 in case of set comparison queries, and type-SCONUCQ1 in case of set containment queries and when it contains a WHERE clause, we call the query of type-SCOMUCQ2 in case of set comparison queries, and type-SCONUCQ2 in case of set containment queries.

5- Processing Set Comparison Correlated Queries

In this section, we give procedures to convert both types of set comparison correlated set queries into either nested queries or unnested queries. The naive way to process such a correlated query is done in two steps. First, the relation in the inner block is scanned for each V value from the outer relation to find its equivalent U value. Second, we need to satisfy the second condition, that is the set of K values associated with this U value is (=/<) the set specified in the WHERE clause of the outer block. This approach has several disadvantages. First, the inner relation should be scanned n times, where n is the number of tuples in the outer relation. Second, query semantics can help, as we see later, to simplify the query. Third, as consequent of second, the query may have a null answer, however, this cannot be recognized before executing the query.

5-1 Processing Type-SCOMCQ1 Queries

This query is written as follows:

```
SELECT RI.C
FROM RI
WHERE "Constant" {=|>} ALL
        SELECT RJ.K
        FROM RJ
        WHERE RJ.U = RI.V
```

In this type of queries, since the outer block contains a constant in its WHERE clause, we need to ensure that the result of the inner block is also a single element, in case of "=ALL". In case of ">ALL", we ensure that all values in the set produced by the inner block are less than "Constant". We transform the query into either a type-N nested query or an unnested query in some case. It is possible to process the whole query by only processing the inner block when the query satisfies the following assumption: Assumption 1: RI.V is the query output and RJ.U $\subseteq$ RI.V.

We have the following possibilities of function dependencies between RJ.U and RJ.K:

1) RJ.U $\not\rightarrow$ RJ.K, 2) RJ.U $\rightarrow$ RJ.K & RJ.K $\rightarrow$ RJ.U, 3) RJ.U $\rightarrow$ RJ.K & RJ.K $\not\rightarrow$ RJ.U.

<u>Case (1): RJ.U $\not\rightarrow$ RJ.K</u>: In this case, the query has the form: Single value {=|>} ALL Set of values. We have the following procedure to process the query. In case of "=ALL" it rejects each U-value in relation RJ, such that the set of K-values associated with this U-value has size greater than one, and for each retained U-value, it deletes it, if its associated K-value is not equal to "Constant". In case of ">ALL", it keeps for each U-value the maximum among values in the set of K-values associated with it, if that maximum is greater than "Constant", its associated U-value is rejected.

(1) Project RJ on U,K to get RJ1 and apply the group-by operation on attribute U of RJ1.

(2) In case of "=ALL", delete all groups with different K-values; retain the other groups by only one normalized tuple representing the group. In case of ">ALL", for each group keep the maximum K-value in the group. Now, we have a relation, called temp with attributes U,K.

(3) For each tuple t in temp, in case of "=ALL", if t[K] = "Constant", retain the tuple, otherwise delete it. In case of ">ALL", delete the tuple if

t[K] > "Constant". The query becomes as follows:

```
        SELECT RI.C
        FROM RI
        WHERE RI.V = SELECT temp.U
                            FROM temp
```

If assumption 1 is satisfied, output temp[U].

Case (2): RJ.U $\rightarrow$ RJ.K & RJ.K $\rightarrow$ RJ.U: There is one-to-one correspondence between U and K values in RJ. In case of "=ALL" the output of the inner block is at most one value. We process the query as follows: Search the inner relation RJ for that tuple t with K value = "Constant", and write the query as follows:

```
SELECT RI.C    FROM RI
WHERE RI.V = "t[U]"
```

If assumption 1 is satisfied, output the value t[U].

In case of ">ALL", it is possible that there are more than one tuple with K-value less than "Constant", we have the following step: Delete from RJ all tuples with K-value > "Constant", having new relation temp. The query becomes as follows:

```
        SELECT RI.C
        FROM RI
        WHERE RI.V = SELECT temp.U
                         FROM temp
```

If assumption 1 is satisfied, output temp[U].

Case (3): RJ.U $\rightarrow$ RJ.K & RJ.K $\not\rightarrow$ RJ.U: From the dependency, we conclude that associated with any U-value at most one K-value. The query is of form: Single value {=|>} Single value. Thus it is of type-J; the following procedure converts it into type-N query.

(1) Project RJ on U,K to get RJ1 and apply the group-by operation on attribute K of RJ1.

(2) In case of "=ALL", delete groups with K-value ≠ "Constant" and in case of ">ALL", delete groups with K-value > "Constant". The query becomes:

```
        SELECT RI.C
        FROM RI
        WHERE RI.V = SELECT temp.U
                            FROM temp
```

If assumption 1 is satisfied, output temp[U].

5-2 Processing Type-SCOMCQ2 Queries

The query has the form:

```
    SELECT RI.C
    FROM RI
    WHERE RI.K {=/>}ALL
                  SELECT RJ.K
                  FROM RJ
                  WHERE RJ.U = RI.V
```

In case of type-SCOMCQ1, the query has a form: Single value (=/>) Set of values. The procedure suggested to process such queries, checks whether the right-hand side set consists of only one value or not. In this case, since the WHERE clause of the outer block contains an attribute, instead of a constant, we have the following possibilities, depending on whether the outer block and the inner block return a single or a set of values:

(1) Single value (=/>) Set
(2) Set (=/>) Single value
(3) Single value (=/>) Single value
(4) Set (=/>) Set

These situations are produced depending on fd's between attributes in the both relations.

Case (1): Single value (=/>) Set of values: We have this form in case of the following dependencies: RI.V $\rightarrow$ RI.K & RJ.U $\not\rightarrow$ RJ.K. That is, for each V-value in the inner block, we are expecting at most one RI.K value. We have the following procedure, which converts the query into a type-J query. In case of "=ALL", it keeps each tuple in RJ, if the set of K-values associated with the U-value of the tuple is of size one. In case of ">ALL", it keeps for each U-value the maximum K-value in the set of K-values associated with this U-value.

(1) Project RJ on U and K and unnormalize the projection on U.

(2) Delete groups of different K-values, in case of =ALL. In case of >ALL, for each group retain the maximum K-value, and delete the others.

(3) Renormalize the relation to be temp(U,K).

(4) The query becomes a type-J query as follows:

```
SELECT RI.C
FROM RI
WHERE RI.K {=/>}
        SELECT temp.K
        FROM temp
        WHERE temp.U = RI.V
```

Case (2): Set of values {=/>} Single value: It happens when the following dependencies are satisfied: RI.V $\not\rightarrow$ RI.K & RJ.U $\rightarrow$ RJ.K. In this case, we apply the previous procedure on RI instead of RJ.

(1) Project RI on V,C, and K and unnormalize the projection on V.

(2) Delete groups of different K-values, in case of =ALL. In case of >ALL, for each group retain the minimum K-value, and delete the others.

(3) Renormalize the relation to be temp(V,K,C). If for some V-value there are more than one associated C-value, we need to memorize all such C-values.

(4) The query becomes a type-J query as follows:

```
SELECT temp.C
FROM temp
WHERE temp.K {=/>}
          SELECT RJ.K
          FROM RJ
          WHERE RJ.U = temp.V
```

Note that, in this case, even if RJ.U is a subset of RI.V, we cannot remove the condition in the WHERE clause of the inner block to convert the query into a type-N query.

Case (3): Single value {=/>} Single value: This is true, when RI.V$\rightarrow$RI.K & RJ.U$\rightarrow$RJ.K. The query is of type-J, and is written by replacing =ALL by =, and >ALL by >.

Case (4): Set of values {=/>} Set of values: In case of =, there is no procedure to convert this query into either a nested query without a set operation or an unnested query. It should be processed by the naive approach. Before applying the second step of the naive approach, we can do preprocess the sets to be compared before actually comparing their elements. First, we check that the two sets of same size. Second, sort the two sets to discover differences as early as possible.

In case of ">", the query has the following semantics; any element e_i in the left-hand side set is greater than any element e_j in the right-hand side set. From this interpretation of the query, we can convert it into a type-J query by applying the following procedure:

(1) Project relation RI on attributes V,K,C, and applying the group-by operation on attribute V.

(2) For each group, keep the minimum K-value among all K-values in the group.

(3) Renormalize the relation, to have a new relation called templ(C,V,K).

(4) Project relation RJ on attributes U,K, and applying the group-by operation on attribute U.

(5) For each group, keep the maximum K-value among all K-values in the group.

(6) Renormalize the relation, to have a new relation called temp2(U,K).

(7) Rewrite the query using relations templ and temp2 as follows:

```
SELECT templ.C
FROM templ
WHERE templ.K >
          SELECT temp2.K
          FROM temp2
          WHERE temp2.U = templ.V
```

6- Processing Set Comparison Uncorrelated Queries

The query has the form:

```
SELECT RI.C
FROM RI
WHERE RI.K {=/>} ALL
                SELECT RJ.K
                FROM RJ
                [WHERE RJ.U = "Const."]
```

6-1 Processing Type-SCOMUCQ1 Queries

The case when the inner block does not contain a WHERE clause. This query has the following semantics: Output C values in RI, if the set of RI.K values (value) associated with this C value is equivalent to the set of values in RJ.K. We have the following cases:

Case (1): RI.C→RI.K & RJ.K→RJ (i.e. RJ.K is the key of RJ): We know that with each RI.C value there is only one associated RI.K value. Also the set of values returned by the inner block is of size greater than one (assuming that RJ has more than one tuple). Thus in case of =ALL the query has a null answer.

Case (2): RI.C→RI.K & RJ.K ↛ RJ: We need to ensure that all the RJ.K values are similar. Process the inner block and if the result is a single value, substitute this value in the original block and replace =ALL by =. The query becomes an unnested. Otherwise the query has a null answer.

In both cases (1) and (2), if the operation is >ALL, it is replaced by the following type-A query:

```
SELECT RI.C
FROM RI
WHERE RI.K        >    SELECT MAX(RJ.K)
                       FROM RJ
```

Case (3): RI.C ↛ RI.K & RJ.K → RJ: In this case, if the number of tuples in relation RJ is n, then the set resulted from the inner block is of size n. in case of =ALL apply the following procedure:

(1) Project RI on C and K to get RI' and unnormalize RI' on C.

(2) Delete all groups with the number of K-values not equal n.

(3) Sort each group on K-values.

(4) Compare each group with the output of the inner block.

In case of >ALL, we process the query as follows:

(1) Replace the inner block by the query:

```
SELECT MAX(RJ.K)        FROM RJ
```

(2) Process this query to have MAXRJK.

(3) Project RI on C and K to get RI' and unnormalize RI' on C.

(4) For each group keep the minimum k-value, and delete all the others. Now, a normalized relation called temp is generated.

(5) The query becomes an unnested query written as follows:

```
SELECT temp.C
FROM temp
WHERE temp.K > "MAXRJK"
```

Case (4): RI.C ↛ RI.K & RJ.K ↛ RJ: In this case the query will be processed by the naive approach.

6-2 Processing Type-SCOMUCQ2 Queries

It is the case when the inner block contains a WHERE clause.We have the following cases:

Case (1): Single value =/> Set of value and Single value =/> Single value: It happens in case of the fd RI.C→RI.K. We process the inner block and if it returns only one K-value as a result,we replace that block by this value and also replace =ALL by =. The query is an unnested query. If it does not, the answer is null. In case of >ALL, we can replace it by type-A query as in cases (1) and (2) above.

Case (2): Set of values =/> Single value: It happens in case of the dependencies RI.C ↛ RI.K & RJ.U → RJ.K are satisfied. In this case the output of the inner block is a single value, while the set corresponds to the outer block may be of size greater than one. We apply the following procedure:

(1) Process the inner block whose result is a single value, say, "KJ".

(2) Project RI on attributes C and K to get RI' and unnormalize RI' on C.

(3) Delete all groups with different K-values.

(4) Output C-values of groups with K-value = "KJ".

In case of >ALL, we keep for each C-value in relation RI' the minimum among all k-values associated with this C-value. The query is converted to the following unnested query:

```
SELECT RI'.C
 FROM RI'
 WHERE RI'.K        >'KJ'.
```

Case (3): RI.C ↛ RI.K & RJ.U ↛ RJ.K : In this case the query is processed by the naive way.

7- Processing Set Containment Queries

A set containment query has the general form: Set 1 IS IN (IS NOT IN) Set 2. Set containment queries are classified into two main classes. Correlated and uncorrelated. Each class is, again, divided into two subclasses. The next section discusses processing correlated queries. Section 7-2 discusses processing uncorrelated queries.

7-1 Processing Set Containment Correlated Queries

A set containment correlated query in SQL is written as follows:

```
SELECT RI.C
FROM RI
 WHERE {"Constant"/ RI.K} IS IN / IS NOT IN
                    SELECT RJ.K
                    FROM RJ
                    WHERE RJ.U = RI.V
```

7-1-1 Processing Type-SCONCQ1 Queries

This the case when the WHERE clause of the outer block contains a constant. In this case Set 1 consists of only one element and its value is specified explicitly by the user. Hence, IS IN can be replaced by =, and the query becomes of type-J, the following procedure, however, converts the query into type-N nested query:

(1) Project RJ on attributes K and U to get a new relation RJ1, and apply Group-by (RJ1,U).

(2) Delete each group consists of only one K-value but not equal to "Constant".

(3) For groups consist of more than one K-value, sort the K-values, and if there is some k-value which is equal to "Constant" keep the group; otherwise delete it. The new relation is RJ2.

(4) Project RJ2 on U to have temp.

(5) The query then can be written in the following form:

```
   SELECT RI.C
   FROM RI
   WHERE RI.V =   SELECT temp.U
                      FROM temp.
```

If assumption 1 is satisfied, then output temp.U.

In case of IS NOT IN, we apply a similar procedure. We keep a group if it contains only one K-value such that that value is not equal to "Constant", or if it consists of more than one K-value and there is no k-value = "Constant".

Note that, if RJ.U → RJ.K, we are sure that Set 2 is of size one, so we can replace IS IN and IS NOT IN by = and ≠, respectively. The query becomes of type-J. We can convert the query into type-N by selecting tuples from relation RJ with K-value equal (not equal) "Constant" in case of IS IN (IS NOT IN) to have relation temp, and write the query as in step 6 above. Note that, if the dependency RJ.K → RJ.U is satisfied, the query can be written as an unnested query.

7-1-2 Processing Type-SCONCQ2 Queries

This case happens when the WHERE clause of the outer block contains an attribute. In this case, the query is of form, Set 1 IS IN / IS NOT IN Set 2. We have similar four possibilities.

(1) Single (IS IN / IS NOT IN) Set

(2) Set (IS IN / IS NOT IN) Single

(3) Single (IS IN / IS NOT IN) Single

(4) Set (IS IN / IS NOT IN) Set

<u>Case (1): Happens when RI.V → RI.K & RJ.U ↛ RJ.K</u>: We can replace the query by a type-J query in case of IS IN by replacing IS IN with =. In case of IS NOT IN the query is processed by the naive way, since RI.K value is not a constant.

<u>Case (2): Happens when RI.V ↛ RI.K & RJ.U → RJ.K</u>: In case of IS IN, apply the following procedure which converts the query into type-J query:

(1) Project RI on attributes C,V, and K to get a new relation RI1.

(2) Apply Group-by (RI1,V), to have relation RI2.

(3) Delete each group consists of diferent K-values. The new relation is RI3.

(4) Renormalize RI3 to have temp.

(5) The query becomes a type-J, and is written as follows:

```
SELECT temp.C
FROM temp
WHERE temp.K = SELECT RJ.K
                    FROM RJ
                    WHERE RJ.U = temp.V
```

Note that, we cannot apply assumption 1.

In case of IS NOT IN, apply the above procedure with modifying step 3 to be:
Divide relation RI2 horizontally into two parts:
1- temp relation as step 3 in the above procedure, and
2- temp2 relation as the rest of relation RI2. That is, temp2 consists of tuples having V-values associated with different K-values.
The output C is obtained as the union of results of two queries:

```
SELECT temp.C
FROM temp
WHERE temp.K ≠ SELECT RJ.K
               FROM RJ
               WHERE RJ.U = temp.V
```

and

```
SELECT temp2.C
FROM temp2
WHERE temp2.V = SELECT RJ.U   FROM RJ
```

<u>Case (3): Happens when RI.V → RI.K & RJ.U → RJ.K</u>: It is replaced by a type-J query. IS IN is replaced by =, and IS NOT IN by ≠.

<u>Case (4): Happens when RI.V ↛ RI.K & RJ.U ↛ RJ.K</u>: We cannot convert the query into a non-set query. We can do two preprocessing steps before actually comparing the sets for containment. (1) Check sizes of the sets. If size of set 1 is greater than size of set 2, set 1 cannot be in set 2. (2) Sort the two sets to minimize comparison time.

7-2 Processing Set Containment Uncorrelated Queries

The query has the form:

```
SELECT RI.C
FROM RI
WHERE RI.K  IS IN / IS NOT IN
        SELECT RJ.K
        FROM RJ
        [WHERE RJ.U = "Constant"]
```

7-2-1 Processing Type-SCONUCQ1 Queries

When the inner block does not contain a WHERE clause. There are four possibilities as before.

<u>Case (1): Single value (IS IN / IS NOT IN) Set of values</u>: It arises when functional dependencies RI.C → RI.K is satisfied and the projection of relation RJ on attribute K consists of more than one value. In case of IS IN the query is of type-N and IS IN is replaced by =. In case of IS NOT IN, we cannot convert the query into a non-set query.

<u>Case (2): Set of values (IS IN / IS NOT IN) Single value</u>: It arises when the projection of relation RJ on K consists of only one value and the fd RI.C ↛ RI.K is satisfied. We apply the following procedure:
(1) Project RI on attributes C and K to get a new relation RI1, and apply Group-by (RI1,C).
(2) Delete groups consist of more than one K-value, in case of IS IN. The new relation is temp.
(3) The query becomes:

```
SELECT temp.C
FROM temp
WHERE temp.K = SELECT RJ.K FROM RJ
```

In case of IS NOT IN, we need to memorize C values with more than one K-value. The result consists of these C values union the result of query similar to the one in step 3 with ≠ replacing =.

<u>Case (3): Single value (IS IN / IS NOT IN) Single value:</u> It happens when the projection of relation RJ on attribute K consists of only one value and the dependency RI.C → RI.K is satisfied. We process the inner query, and then replace it by the only k-value generated. The query becomes unnested, IS IN is replaced by =, and IS NOT IN by ≠.

<u>Case (4): Set of values (IS IN / IS NOT IN) Set of values</u>: The query is processed by the naive way.

7-2-2 Processing Type-SCONUCQ2 Queries

When the inner block contains a WHERE clause. There are the following cases:

Case (1): Single value IS IN/IS NOT IN Set of values: It happens in case of the function dependencies RI.C $\rightarrow$ RI.K & RJ.U $\not\rightarrow$ RJ.K. Since Set 1 is of size one and Set 2 is of size greater than one, we can replace IS IN by =. IS NOT IN cannot be replaced by $\neq$, and the query will be processed first by processing the inner block and check that RI.K is not in the generated set.

Case (2): Single value IS IN/IS NOT IN Single value: It happens in case of the function dependencies RI.C $\rightarrow$ RI.K & RJ.U $\rightarrow$ RJ.K.We can replace IS IN by = and IS NOT IN by $\neq$. The query can be converted into an unnested query, by substituting the inner block with its result.

Case (3): Set of values IS IN / IS NOT IN Single value: It happens in case of the function dependencies RI.C $\not\rightarrow$ RI.K & RJ.U $\rightarrow$ RJ.K.We apply the following procedure in case of IS IN:

(1) Process the inner block to obtain its result, say RJK.

(2) Project relation RI on attributes C and K, and apply group-by on attribute C.

(3) Delete all groups with different K-values. For retained tuples, delete any one with K $\neq$RJK.

(4) Output the remaining C-values.

In case of IS NOT IN, apply the first three steps, and then delete all groups with a K-value = RJK. Output the retained C-values.

Case (4): Set of values IS IN / IS NOT IN Set of values: It happens in case of the function dependencies RI.C $\not\rightarrow$ RI.K & RJ.U $\not\rightarrow$ RJ.K. In this case we apply the naive approach.

8- Conclusion

In this paper, we studied optimizing set oriented SQL queries. That is queries that compare between sets of values. We optimize such queries, by converting the query into a non-set query. Almost all set queries, except only one type, can be converted into either type-N, type-J, type-A nested or in some cases into an unnested query.

References

[1] Astrahan,M.M, et al., "System R: Relational Approach to Database Management", ACM TODS, Vol.1, No.2, June 1976, pp.97-137.

[2] Bernstein,P., Chiu,W.,"Using Semi-Joins to Solve Relational Queries", J. ACM, Vol.28, No. 1, pp.25-40.

[3] Chamberlin,D.D., et al., "SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control", IBM J. Res. Dev. Nov.1976, pp.560-575.

[4] Codd,E.F. "A Relational Model for Large Shared Data Banks", Comm. ACM, Vol.13, No.6, pp.377-387.

[5] Date,C.J. An Introduction to Database Systems, 3rd ed., Addison-Wesely, 1981.

[6] Goodman,N., Shmueli,O. "Tree Queries: A Simple Class of Relational Queries", ACM TODS, Vol.7, No.4, Dec.1982, pp.653-677.

[7] Jark,M., Koch,J., "Query Optimization in Database Systems", ACM Comp. Surveys, Vol.16, No.2, June 1984, pp.111-151.

[8] Kambayashi,Y., Yoshikawa,M., Yajima,S. " Query Processing Utilizing Dependencies and Horizontal Decomposition", ACM SIGMOD, May 1983, pp.55-67.

[9] Kim,W. "On Optimizing an SQL-like Nested Query", ACM TODS, Vol.7, No.3, pp.443-469.

[10] Ullman,D.J., Principles of Database Systems, 2ed edition, Computer Science Press, 1982.