IMPROVED QUERY PLANS FOR UNNESTING SQL NESTED QUERIES

THESIS REPORT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Computer Science and Engineering

by

SATHISH KUMAR M

Department of Computer Science and Engineering
National Institute of Technology
Rourkela

2007

IMPROVED QUERY PLANS FOR UNNESTING SQL NESTED QUERIES

THESIS REPORT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Computer Science and Engineering

by

SATHISH KUMAR M

Under the guidance of

Prof. S. K. JENA

Department of Computer Science and Engineering
National Institute of Technology
Rourkela

2007

National Institute of Technology

Rourkela

CERTIFICATE

This is to certify the thesis entitled, *Improved Query plans for Unnesting SQL Nested Queries* submitted by Sri.Sathish Kumar M in partial fulfillment of the requirements for the award of Master of Technology Degree in Computer Science and Engineering at National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Prof. S. K. Jena
Dept. of Computer Science and Engg.
National Institute of Technology
Rourkela - 769008

Date:

Acknowledgements

I take this opportunity to express my gratitude towards my thesis guide Prof. S. K. Jena for showing the way and guiding me constantly at all stages, while working on this thesis. In fact the sound understanding of the problems and clearer communication of ideas and suggestions by him can said to have done much of this thesis work.

I also extend my sincere thanks to all Professors in Computer Science and Engineering department for all the help and support during this thesis. I would also like to give special thanks to my batch mates and other friends in the institute, who by way of their little advice and suggestions. Many of their suggestions have found a way in the final form of the thesis also.

At last, but not the least, I deeply appreciate all my family members and all my cousins for their emotional support during the entire course work.

Sathish Kumar M
(20506013)
May 2007

Contents

Abstract

The SQL language allows users to express queries that have nested subqueries in them. Optimization of nested queries has received considerable attention over the last few years. The first algorithm for unnesting nested queries was Kim's algorithm, but this technique had a COUNT bug for JA type queries. Later few researchers gave more general strategies to avoid the COUNT bug. Finally to all this M. Muralikrishna modified Kim's algorithm so that it avoids the COUNT bug. The modified algorithm may be used when it is more efficient than the general strategy. In addition, he presented a couple of enhancements that pre-compute aggregates and evaluate joins and outer joins in a top down order. These enhancements eliminated Cartesian products when certain correlation predicates are absent and enabled us to employ Kim's method for more blocks. Apart from this he proposed the Integrated algorithm for generating query plans for a given input query.

In this thesis we have given a new solution for implementing the Kim's modified algorithm of unnesting nested queries and this also avoids the COUNT bug convincingly. Integrated algorithm generates flaws query plans, which has been modified in this thesis. We have also shown experimental results proving one query plan among the all other as computationally better one. These computations are in terms of elapsed time. We have carried out experiments for different data sets of varying sizes from 100 to 1000 tuples in each relation. These results are taken as average of some possible iterative execution of each query plan. Finally, we incorporate the above improved merits into a new unnesting algorithm.

List of Figures

List of Tables

Chapter 1

Introduction to Nested Queries

SQL is a block-structured query language for data retrieval and manipulation developed at the IBM Research Laboratory in San Jose, California [1] SQL was incorporated into System R, the relational data base management system, also developed at the IBM San Jose Research Laboratory [2]. One of the most powerful features of SQL is the nesting of query blocks. Traditionally, database systems have executed nested SQL [1] queries using Tuple Iteration Semantics (TIS). It was analytically shown in [8] that executing queries by TIS can be very inefficient. It was first pointed out in [5] and then in [8] that nested queries can be evaluated very efficiently using relational algebra or set-oriented operators. "The process of obtaining set-oriented operators to evaluate nested queries is known as **unnesting**".

It was later pointed out in [7] and [6] that the unnesting techniques presented in [8] do not always yield the correct results for nested queries that have non equi-join correlation predicates or for queries that have the COUNT aggregate between nested blocks. Unnesting solutions for these types of queries were provided in [6]. These solutions were further refined and extended in [4]. An important contribution of the current thesis is a successful implementation for Kim's modified algorithm that avoids the COUNT bug. Under certain conditions, Kim's approach may be more efficient than the general solution and hence worth considering.

In this thesis, we focus our attention on unnesting Join-Aggregate (JA) type of SQL queries [8]. These queries have correlation join predicates and an aggregate (AVG, SUM, MIN, MAX, or COUNT) between the nested blocks. The reason for focusing on JA type queries is that many other nesting predicates (such as EXISTS, NOT EXISTS, ALL, ANY) can be reduced to JA type queries [6], [4].

An example of a 2 block JA type query is:

```
SELECT R1.a
FROM   R1
WHERE  F1(R1)
AND    R1.b OP1 (SELECT COUNT (R2.*)
                 FROM   R2
                 WHERE  F2(R2) AND F2(R2,R1))
```

F1(R1) and F2(R2) are selection predicates on R1 and R2 respectively, while F2(R2, R1) is a correlation join predicate between R1 and R2.

A run time system that would execute the above query using TIS would proceed as follows: A tuple r1 from R1 would be fetched. If F1(R1) is false for r1, tuple r1 will not be present in the result. Assuming F1(R1) is true, the values of the relevant attributes of r1 would be substituted into predicates at deeper levels (F2(R2, R1)). The two block query now becomes a single block query

```
SELECT COUNT (R2.*)
FROM R2
WHERE F2'(R2)
```

F2'(R2) is a predicate on R2 and is equivalent to F2(R2) AND F2(R2, R1) after values of r1's attributes have been substituted in F2(R2, R1).

Let the COUNT value returned by this block be C ($C \leq 0$). C represents the number of tuples of R2 that satisfy F2'(R2). If (r1.b OP1 C) is true, rl will be in the result. Notice that each tuple of R1 can occur in the result at most once. Using TIS, the system executes a query on R2 (the inner relation) for every tuple of R1 (the outer relation) leading to a very inefficient execution strategy [8]. Blocks in the above nested query may be nested within each other to any arbitrary depth.

There are two types of nested queries and they are defined as follows:

Nested Linear Query: is a JA type query in which at most one block is nested within any block.

Nested Tree Query: is a JA type query in which there is at least one block which has two or more blocks nested within it at the same level.

In this thesis, we focus our attention on linear queries only.The techniques for unnesting tree queries presented in [10] were not as general as the ones we are developing in the current thesis. For example, [10] did not consider Kim's algorithm at all. For ease of notation, we shall assume that there is only one relation in the FROM clause of each block. The algorithms presented in this thesis can be easily extended to the case when there are multiple relations in any FROM clause.

The reader is advised that we shall not adhere to strict SQL syntax when writing queries in this thesis. The SQL syntax for expressing outer joins is fairly cumbersome. Instead, we shall write queries in a syntax that is fairly intuitive.

Chapter 2

Types of Nested Queries

Won Kim developed a classification of nested query types, four of which are relevant to this thesis. They are described here briefly for single-level nested queries, as presented in [8].

2.1 Type-A Nesting

A nested predicate is type-A if the inner query block Q does not contain a Join predicate that references a relation in the outer query block, and if the SELECT clause of Q consists of an aggregate function over a column in an inner relation [8]. The following is an example of a type-A nested query of depth one.

```
SELECT SNO
FROM   SP
WHERE  PNO = (SELECT MAX(PNO)
              FROM    P)
```

Since the inner query block of a type-A nested query does not reference a relation of the outer query block. It may be evaluated independently of the outer query block, and the result of its evaluation will be a single constant.

2.2 Type-N Nesting

A nested predicate is type-N if the inner query block Q does not contain a join predicate which references a relation m the outer block, and the SELECT clause of Q does not contain an aggregate function [8]. The following is an example of a type-N nested query.

Evaluation of a Type-N Nested Query. This kind of nested query would be processed in System R by first processing the inner query block Q, resulting in a list of values X which can then be substituted for the inner query block in the nested predicate, so that PNO IS IN Q becomes PNO IS IN X The resulting query is then evaluated by nested iteration.

2.3 Type-J Nesting

A type-J nested predicate results when the WHERE clause of the inner query block contains a join predicate which references the relation of an outer query block, and the relation is not mentioned in the inner FROM clause. Another condition is that the SELECT clause of the inner query block does not contain an aggregate function [8]. The following is an example of type-J nesting.

```
SELECT SNAME
FROM   S
WHERE  SNO IS IN (SELECT SNO
                  FROM   SP
                  WHERE  QTY > 100 AND
                         SP.ORIGIN = S.CITY)
```

2.4 Type-JA Nesting

Type-JA nesting is present when the WHERE clause of the inner query block contains a join predicate which references the relation of an outer query block, and the inner SELECT clause consists of an aggregate function over an inner relation [8]. Select names of parts which have the highest part number in the city from which they are supplied.

```
SELECT PNAME
FROM   P
WHERE  PNO = (SELECT MAX(PNO)
              FROM   SP
              WHERE  SP.ORIGIN = P.CITY)
```

2.5 Evaluation of Type-J and Type-JA Nested Queries

Type-J and type-JA nesting are processed in System R by the nested iteration method the inner query block is processed once for each tuple of the outer relation which satisfies all simple predicates on the outer relation. This method has the obvious disadvantage that the inner relation (SP in above example) may have to be retrieved many times. It must be retrieved once for each tuple of the outer relation S, since there are no simple predicates in the outer query block. It is this inefficiency which motivated Kim to develop alternative algorithms for processing nested queries.

Chapter 3

A Review on Unnesting Nested Queries

3.1 Kim's Algorithms for Processing Nested Queries

Kim observed that for type-N and type-J nested queries, the nested iteration method for processing nested queries is equivalent to performing a join between the outer and inner relations [8]. But nested iteration is only one way of performing a join, for single-level queries System R also performs joins by the merge join method, with the decision as to which method to use made by the query optimizer. Kim showed that nested queries could be transformed to logically equivalent single-level queries containing single-level join predicates explicitly, and that now the query optimizer can choose a merge join method in implementing the joins, often at a great reduction of cost over the nested iteration method [8]. Kim's transformation algorithms are summarized in the present chapter.

3.1.1 Processing a Type-N or Type-J Nested Query

In his Lemma 1 [8], Kim states that a type-N nested two-relation query is equivalent to a canonical two-relation query with a join predicate.
Let Q1 be

```
SELECT Ri.Ck
FROM   Ri,Rj
WHERE  Ri.Ch = Rj.Cm
```

and Let Q2 be

```
SELECT Ri.Ck
FROM   Ri
WHERE  Ri.Ch IS IN (SELECT Rj.Cm
                    FROM   Rj)
```

Kim's Lemma 1 states that Ql and Q2 are equivalent, that is, they yield the same result [8]. Kim's proof of lemma 1 calls attention to the fact that by definition the inner block of Q2 can be evaluated independently of the outer block, resulting in a list of values. Since this list contains values from column

Rj.Cm, the predicate is equivalent to the join predicate Ri.Ch = Rj.Cm [8]. From Lemma 1 Kim develops the following algorithm

Algorithm NEST-N-J

1. Combine the FROM clauses of all query blocks into one FROM clause
2. AND together the WHERE clauses of all query blocks, replacing IS IN by =
3. Retain the SELECT clause of the outermost query block

The result is a canonical query logically equivalent to the original nested query. The algorithm applies to type-N or type-J nested queries with one or more levels of nesting.

3.1.2 Processing a Type-JA Nested Query

In his Lemma 2 [8], Kim asserts that a type-JA nested query can be transformed to a type-J nested query which references a new temporary relation

Let Q3 be

```
SELECT Ri.Ck
FROM   Ri
WHERE  Ri.Ch = (SELECT AGG(Rj.Cm)
                FROM   Rj
                WHERE  Rj.Cn = Ri.Cp)
```

and Let Q4 be

```
SELECT Ri.Ck
FROM   Ri
WHERE  Ri.Ch = (SELECT Rt.C2
                FROM   Rt
                WHERE  Rt.C1 = Ri.Cp)
```

where Rt is a temporary table obtained by

```
Rt(C1,C2)= (SELECT    Rj.Cn, AGG(Rj.Cm)
            FROM      Rj
            GROUP BY  Rj.Cn)
```

Kims Lemma 2 states that Q3 and Q4 are equivalent [8]. His proof postulates that the action of the nested iteration processing of a type-JA query can be captured in a temporary table formed with a GROUP BY clause, as in Rt for each tuple of Ri, a tuple is retrieved from Rt whose C1 (formerly Cn) value matches the Cp value of the Rt tuple. The C2 value of the Rt tuple will contain the aggregate value obtained by the GROUP BY clause, and this can be matched with Ri.Ch [8].
Lemma 2 leads to an algorithm which transforms a type-JA nested query of depth one to an equivalent type-J nested query of depth one. Assume a type-JA nested query as follows

```
SELECT R1.Cn+2
FROM   R1
WHERE  R1.Cn+1 = (SELECT AGG(R2.Cn+1)
                  FROM   R2
                  WHERE  R2.C1=R1.C1 AND
                         R2.C2=R1.C2 AND


                         R2.Cn = R1.Cn)
```

Algorithm NEST-JA

1. Generate a temporary relation Rt(C1, ,Cn,Cn+l) from R2 such that Rt.Cn+l is the result of applymg the aggregate function AGG on the Cn+l column of R2 which have matching values in R1 for C1, C2, etc

2. Transform the inner query block of the initial query by changing all references to R2 columns in join predicates which also reference R1 to the corresponding Rt columns. The result is a type-J nested query, which can be passed to algorithm NEST-N-J for transformation to its canonical equivalent.

3.2 Bugs in Kim's Algorithm NEST-JA and their Solutions

3.2.1 The COUNT bug

In a 1984 U C Berkeley Memorandum [7], Werner Kiessling revealed a problem with Kim's algorithm NEST-JA. The problem arises when a type-JA nested query contains the COUNT function. To illustrate his arguments, Kiessling defines two relations.

PARTS(PNUM,QOH)
SUPPLY(PNUM,QUAN,SHIPDATE)

The following instantiations of these relations are assumed in figures 3.1 and 3.2.

PNUM	QOH
3	6
10	1
8	0

Table 3.1: PARTS table

PNUM	QUAN	SHIPDATE
3	4	7-3-79
3	2	10-1-78
10	1	6-8-78
10	2	8-10-81
8	5	5-7-83

Table 3.2: SUPPLY table

Kiessling defines Query Q2 as follows:
Query Q2:
Find the part numbers of those parts whose quantities on
hand equal the number of shipments of those parts before I-I-80.

```
SELECT PNUM
FROM   PARTS
WHERE  QOH = (SELECT COUNT(SHIPDATE)
              FROM   SUPPLY
```

```
                    WHERE  SUPPLY.PNUM = PARTS.PNUM AND
                           SHIPDATE < 1-1-80)
```

Given the example tables PARTS and SUPPLY shown in Figures 3.1 and 3.2 , query Q2 will give the following result when evaluated using nested iteration.

RESULT

PARTS.PNUM
10
8

Application of Kim's algorithm NEST-JA to Query Q2 results in the following transformation.

```
TEMP'(SUPPNUM,CT) = (SELECT     PNUM,COUNT(SHIPDATE)
                     FROM       SUPPLY
                     WHERE      SHIPDATE < 1-1-80
                     GROUP BY   PNUM)

SELECT PNUM
FROM   PARTS,TEMP'
WHERE  PARTS.QOH = TEMP'.CT AND
       PARTS.PNUM = TEMP'.SUPPNUM
```

TEMP' evaluates to

SUPPNUM	**CT**
3	2
10	1

and final result is

PARTS.PNUM
10

This result offers from that obtained using nested iteration. The reason why the transformation fails is that in the formation of the temporary relation, no tuples appear which do not match the predicates applied to the inner relation. Thus, the COUNT function will never return zero, since the only groups it is applied to are groups of tuples matching the predicates. Thus CT in the temporary relation will never be zero.

Kiessling explored a trial correction of the bug which involved ORing a predicate to the WHERE clause of the transformed query in order to a posteriori find where an empty set occurs to satisfy the predicate, but the final correction failed on a query with more than one level of nesting [7]. Kiessling concludes that in attempting to use Kim's algorithm NEST-JA for transforming type-JA nested queries, "there seems to be no general way to recover values lost by COUNTS on a correlation level greater than one" [7]. While this does seem to be true in the context of the SQL language as specified in [1], the problem can be solved if the outer join operation is available in the processing of the query.

3.2.2 Solution to the COUNT bug using outer joins

If either internally or through extensions to the query language an outer join operation may be specified as the join operation, the COUNT bug can be solved by performing an outer join in the creation of the temporary relation. The operation of outer join is defined in [3] . The outer join includes all values from columns participating in join with NULLS in the opposite column if there is no match for a column value.

In 1987 Richard A Ganski gave a solution for COUNT bug mentioned above. To solve the COUNT bug an outer join may be used in the creation of the temporary relation Kiessling's query Q2 could be transformed to give the following

```
TEMP3(SUPPNUM,CT) = (SELECT    PARTS.PNUM,COUNT(SHIPDATE)
                     FROM      PARTS P, SUPPLY S
                     WHERE     S.SHIPDATE < 1-1-80 AND
                               P.PNUM =+ S.PNUM
                     GROUP BY  P.PNUM)
```

Query T3

```
SELECT PNUM
FROM   PARTS P, TEMP3
WHERE  P.QOH = TEMP3.CT AND
       P.PNUM = TEMP3.SUPPNUM
```

Before looking at the result of this new query, let us look at the result of the outer join between PARTS and SUPPLY with the conditions given in the creation of the temporary relation TEMP3 shown in Figure 3.3

P.PNUM	P.QOH	S.PNUM	S.QUAN	S.SHIPDATE
3	6	3	4	7-3-79
3	6	3	2	10-1-78
10	1	10	1	6-8-78
8	0	NULL	NULL	NULL

Table 3.3: PARTS LEFT OUTER JOIN SUPPLY

Note that the condition which applies to only one relation (SUPPLY.SHIPDATE < I-I-80) must be applied before the join is performed. Otherwise the join would not contain the last row, and the result would be incorrect. This may happen if the join is performed first to take advantage of indices on the join columns. To ensure restriction, we can explicitly build a temporary table applying simple predicates. This temporary table will be a restriction and projection of the inner table.

```
TEMP2(PNUM) (SELECT PNUM
             FROM   SUPPLY
             WHERE  SHIPDATE < 1-1-80)
```

and TEMP3 changed to

```
TEMP3(SUPPNUM,CT) = (SELECT    PARTS.PNUM,COUNT(TEMP2.SHIPDATE)
                     FROM      PARTS P, TEMP2
                     WHERE     P.PNUM =+ TEMP2.PNUM
                     GROUP BY  P.PNUM)
```

Thus, TEMP3 will look like this

SUPPNUM	**CT**
3	2
10	1
8	0

Table 3.4: TEMP3

and the result of T3 will be

PARTS.PNUM
10
8

which matches the result obtained by nested iteration. This solution has been tested successfully on queries with more than a single level of nesting.

If the type-JA query with a COUNT function contains a nested join predicate with a scalar comparison operator other than equality, the correct result is obtained, if the scalar operator is used in the outer join operation to create the temporary relation and the join predicate in the original query is changed to equality.

3.2.3 Query Blocks with COUNT(*)

If the SELECT clause of the inner query block contains COUNT(*) instead of COUNT(column name) then this approach must be modified. For example, if query Q2 contained a COUNT(*) instead of a COUNT(SHIPDATE), then the temporary table would look like this

SUPPNUM	CT
3	2
10	1
8	1

Table 3.5: TEMP3 with COUNT(*)

This would be semantically incorrect, and the final result would be incorrect. To avoid this error the SELECT clause used in the creation of the table must contain COUNT(col-name) instead of COUNT(*), where col-name is the name of some column in the inner relation. Since the join column of the inner relation will always be present in the original query and may be the only one that is, let col-name be the name of the join column of the inner relation. In our example it would be COUNT(TEMP2.PNUM).

3.2.4 A Problem with Relations other than Equality

For aggregate functions other than COUNT, Kim's algorithm NEST-JA works correctly for nested join predicates containing the equality operator. However, If we consider other operators, we discover another bug in Kim's algorithm.
Assume the PARTS and SUPPLY tables shown in Figures 3.6 and 3.7

PNUM	QOH
3	0
10	4
8	4

Table 3.6: PARTS table for relations other than equality

PNUM	QUAN	SHIPDATE
3	4	7-3-79
3	2	10-1-78
10	1	6-8-78
9	5	3-2-79

Table 3.7: SUPPLY table for relations other than equality

and the following type-JA query
Query Q5

```
SELECT PNUM
FROM   PARTS
WHERE  QOH = (SELECT MAX(QUAN)
              FROM   SUPPLY
              WHERE  SHIPDATE < PARTS.PNUM AND
                     SHIPDATE < 1-1-80)
```

This is the same as Kiessling's query Q1 [7] except for the substitution of the "$<$" operator for "$=$" operator in the join predicate. The result according to nested iteration semantics, assuming $MAX(\{\}) =$ NULL, is

PARTS.PNUM
8

Kim's algorithm results in the following temporary table and transformed query

```
TEMP5(SUPPNUM,MAXQUAN) = (SELCT     PNUM,MAX(QUAN)
                          FROM      SUPPLY
                          WHERE     SHIPDATE < 1-1-80
                          GROUP BY PNUM
```

Query T5:

```
SELECT PNUM
FROM   PARTS P, TEMP T
WHERE  P.QOH = T.MAXQUAN AND
       T.SUPPNUM < P.PNUM
```

and the following results shown in Figures 3.8 and 3.9.

SUPPNUM	MAXQUAN
3	4
10	1
9	5

Table 3.8: TEMP5

PARTS.PNUM
10
8

Table 3.9: FINAL RESULT with other than equality bug

which does not match the results obtained by nested iteration. The problem is that the temporary table created by Kim's algorithm contains only aggregate information about tuples with the same join column value, whereas query Q5 asks for aggregate information about a range of join column values.This was noticed by the same Richard A Ganski and gave the solution described below.

3.2.5 Solution to the problem with Relations other than Equality

The solution to this bug is similar to the solution to the COUNT bug perform a join in the creation of the temporary relation, only this time it need not be an outer join, unless the aggregate function is COUNT. The join in effect causes the temporary table to include aggregate values over the proper range of join column values. As before, the join predicate in the original query must be changed to equality. This implies that only the equality operator may be the outer relation and the temporary relation.

If this solution is applied to query Q5 and the last SUPPLY table, the outcome is

```
TEMP6(SUPPNUM,MAXQUAN) = SELECT     P.NUM, MAX(S.QUAN)
                         FROM       PARTS P, SUPPLY S
                         WHERE      SHIPDATE < 1-1-80 AND
                                    S.PNUM < P.PNUM
                         GROUP BY P.PNUM
```

and query Q5 is transformed to

Query T6

```
SELECT PNUM
FROM   PARTS P, TEMP T
WHERE  P.QOH = T.MAXQUAN AND
       P.PNUM = T.SUPPNUM
```

with the following results shown in Figures 3.10 and 3.11.

SUPPNUM	MAXQUAN
10	5
8	4

Table 3.10: TEMP6

PARTS.PNUM
8

Table 3.11: FINAL OUTCOME with the solution for other than equality bug

This matches the result obtained by nested iteration.

3.2.6 A Problem with Duplicates

The methods outlined above to solve the COUNT bug work correctly, if the outer relation of the nested query contains no duplicates in the join column, but a problem arises if it does contain duplicates. Assume the following PARTS and SUPPLY relations shown in Figures 3.12 and 3.13.

PNUM	QOH
3	6
3	2
10	1
10	0
8	0

Table 3.12: PARTS with duplicates

PNUM	QUAN	SHIPDATE
3	4	8-14-77
3	2	11-11-78
10	1	6-2-76

Table 3.13: SUPPLY with duplicates

For this example let us again assume Kiessling's query Q2 if we apply query Q2 to the above relations, the result by nested iteration would be shown in Figure 3.14.

PARTS.PNUM
3
10
8

Table 3.14: RESULT BY NESTED ITERATION

If we apply our new modified version of Kim's algorithm, the results would be as shown in Figures 3.15 and 3.16.

SUPPNUM	CT
3	4
10	2
8	0

Table 3.15: TEMP3 for Kim's algorithm

PARTS.PNUM
8

Table 3.16: FINAL RESULT with duplicates problem

This does not match the result obtained by nested iteration. The problem arises because duplicates in the outer relation increase the COUNT over that column in the temporary relation. This problem does not arise with the MAX and MIN functions, but it does arise with the COUNT, AVG and SUM functions.This was also noticed by the same Richard A Ganski and gave the solution described below.

3.2.7 Solution to the Duplicates Problem

In order to match the results obtained by nested iteration semantics for relations with duplicates in the outer join column, our algorithm must be modified to remove duplicates before the join in the creation of the temporary table is performed. This can be accomplished by projecting the join column of the outer relation, and using the projection instead of the outer relation in any join required to build a temporary table. The efficiency of the algorithm can be improved by applying all simple predicates to the outer relation in the creation of the projection. In query Q2 this rule will have no effect since there are no simple predicates in the outer query block.

Using Kesslings query Q2 as an example again, let TEMPI be defined as follows

```
TEMP1(PNUM) = (SELECT DISTINCT PNUM
               FROM   PARTS)
```

TEMPI is the projection of the PNUM column from PARTS TEMP3 will now be defined as

```
TEMP3(SUPPNUM,CT) = SELECT     T1.PNUM, COUNT(S.SHIPDATE)
                    FROM       TEMP1 T1, SUPPLY S
                    WHERE      S.SHIPDATE < 1-1-80 AND
                               T1.PNUM = + S.PNUM
                    GROUP BY   T1.PNUM
```

and query T3 remains the same. The results are as shown in Figures 3.17, 3.18 and 3.19.

which matches the result obtained by nested iteration.

PNUM
3
10
8

Table 3.17: TEMP1

SUPPNUM	CT
3	2
10	1
8	0

Table 3.18: TEMP3 with no duplicates problem

PARTS.PNUM
3
10
8

Table 3.19: FINAL OUTCOME

3.2.8 Modified algorithm NEST-JA2

The solutions to the bugs described in the previous section suggest a modified algorithm for transforming type-JA nested queries, which shall be called algorithm NEST-JA2. Thus algorithm consists of three major parts.

Algorithm NEST-JA2

1. Project the join column of the outer relation, and restrict it with any simple predicates applying to the outer relation.

2. Create a temporary relation, joining the inner relation with the projection of the outer relation. If the aggregate function is COUNT, the join must be an outer join, and the inner relation must be restricted and projected before the join is performed. If the aggregate function is COUNT(*), compute the COUNT function over the join column. The join predicate must use the same operator as the join predicate in the original query (except that it must be converted to the corresponding outer operator in the case of COUNT), and the join predicate in the original query must be changed to =. In the SELECT clause, select the join column from the outer table in the join predicate instead of the inner table. The GROUP BY clause will also contain columns from the outer relation.

3. Join the outer relation with the temporary relation, according to the transformed version of the original query.

To illustrate the action of algorithm NEST-JA2, let us apply it to Kiessling's query Q2 The three steps are then as follows

```
TEMP1(PNUM)= SELECT DISTINCT PNUM
             FROM   PARTS,
```

```
TEMP2(PNUM)= (SELECT PNUM
             FROM    SUPPLY
             WHERE   SHIPDATE < 1-1-80),
```

```
TEMP3(PNUM,CT) = (SELECT    T1.PNUM,COUNT(T2.SHIPDATE)
                 FROM       TEMP1 T1,TEMP2 T2
                 WHERE      T1.PNUM = + T2.PNUM
                 GROUP BY   T1.PNUM),
```

```
SELECT PNUM
FROM   PARTS P,TEMP3 T3
WHERE  P.QOH = T3.CT AND
       P.PNUM = T3.PNUM
```

If these three steps are applied to the PARTS and SUPPLY relations with duplicates considered above, the results are shown in Figures 3.20, 3.21 and 3.22. These match with the result obtained by nested iteration.

PNUM
3
10
8

Table 3.20: TEMP1 of Modified NEST-JA2

SUPPNUM	**CT**
3	2
10	1
8	0

Table 3.21: TEMP3 of Modified NEST-JA2

PARTS.PNUM
3
10
8

Table 3.22: FINAL OUTCOME of Modified NEST-JA2

3.3 Linear Queries with Multiple Blocks

The solution in [4] generalizes Ganski's solution for queries with more than 2 blocks. A linear query with multiple blocks gives rise to a ' linear J/OJ expression ' where each instance of an operator is either a join or an outer join. A general linear J/OJ expression would look like:

R1 J/OJ R2 J/OJ R3 J/OJ . . . J/OJ Rn

Relation R1 is associated with the outermost block, relation R2 with the next inner block and so on. An outer join is required if there is a COUNT between the respective blocks. In all other cases (AVG, MAX, MIN, SUM), we need perform only a join. The joins and outer joins are evaluated using the appropriate predicates. Since joins and outer joins do not commute with each other, a legal order may be obtained by computing all the joins first and then computing the outer joins in a left to right order (top to bottom if you like) [4]. Thus, the expression R1 OJ R2 J R3 J R4 OJ R5 J R6 can be legally evaluated as ((RI OJ (R2 J R3 J R4)) OJ (R5 J R6)). Since we can evaluate joins in any order, we can choose the cheapest join order to join R2, R3, and R4.
It is worth pointing out here that the solution presented in [6] for multiple level queries was incomplete in the sense that it does not discuss legal orderings when joins and outer joins are present in the same expression.

After all the joins and outer joins have been evaluated, the aggregate functions are evaluated in a bottom-up order after grouping the result by the appropriate unique keys. This is best illustrated with an example.

Consider the three block linear query Example 1

```
SELECT R1.a
FROM   R1
WHERE  F1(R1)
AND    R1.b OP1 (SELECT COUNT(R2.*)
                 FROM   R2
                 WHERE  F2(R2) AND F2(R2, R1)
                 AND    R2.c OP2 (SELECT (COUNT(R3.*))
                                  FROM    R3
                                  WHERE   F3(R3) AND F3(R3, R2)
                                  AND     F3(R3, R1)))
```

The relation associated with block (or node) i is represented by Ri (i > 0). Lower case letters (a, b, etc.) represent attribute names. A '*' is used to denote all the attributes of a relation. Ri.# is some unique key of Ri. ri, ri', ri" are each used to denote a tuple of relation Ri. OPn (n > 0) is any one of the following operators $(<, \neq, <, \leq, >, \geq)$. Fi(Rj) represents a selection predicate in the ith block on Rj. To simplify the notation, we will assume that all join predicates am binary. A join predicate in the ith block is then represented as Fi(Rj, Rk), where j, k > 0 and j $\neq$ k.

In the above Example F1(R1), F2(R2) and F3(R3) are selection predicates on R1,R2 and R3 respectively, while F2(R2, R1) is a correlation join predicate between R1 and R2, F3(R3, R2) is a correlation join predicate between R3 and R2, F3(R3, R1) is a correlation join predicate between R3 and R1.

The corresponding linear expression is R1 OJ R2 OJ R3 and hence a legal order is (R1 OJ R2) OJ R3. The predicate for R1 OJ R2 is F2(R2, R1) and the predicate for the outer join with R3 is F3(R3, R2) AND F3(R3, R1). We now show how the query of above Example can be evaluated using set-oriented operations. The result is obtained by executing more than one query. The result from one query may be pipelined to the next query. The two queries in this case are (not in strict SQL syntax!):

```
Query A:
SELECT INTO TEMP(R1.#, R1.a, R1.b, R2.*)
FROM R1, R2, R3
WHERE (R1 OJ R2) OJ R3
GROUP BY R1.#, R2.#
HAVING R2.c OP2 COUNT(R3.*)
```

```
Query B:
SELECT R1.a
FROM TEMP
GROUP BY R1.#
HAVING R1.b OP1 COUNT(R2.*)
```

The results from Query A are fed into Query B. Even though the selection predicates (Fi(Ri), i = 1, 2, 3) have not been shown in Query A, they are applied to the respective relations before they participate in the outer joins. The outer join predicates are also implicit in Query A.

3.3.1 A Few Subtleties

In 1989 M. Muralikrishna noticed a few subtleties that were not mentioned in [4] and deserve to be highlighted. The outer join between R1 and R2 results in two sets of tuples, viz., (R1- X NULL) and R1R2. R1R2 denotes the set {(r1, r2): F2(R2) AND F2(R2, R1) AND F1(R1)}, where the r1 tuple ∈ R1 and the r2 tuple ∈ R2. Let R1+ denote the set of tuples of R1 present in R1R2 (tuples of R1 that participated in the join with R2). R1- denotes the set R1 - (R1+) (the tuples of R1 in the anti-join).

Similarly, let R1R2R3 denote the set {(r1, r2, r3): F3(R3) AND F3(R3, R2) AND F3(R3, R1) AND F2(R2) AND F2(R2, R1) AND F1(R1)}. Let the set of (r1, r2) tuples in R1R2 that joined with at least one tuple of R3 be denoted by R1R2+. The set of (r1, r2) tuples that did not join with any tuple of R3 is denoted by R1R2- and is equal to R1R2 - (R1R2+). Thus, the outer join with R3 may yield up to three distinct sets of tuples, viz., (R1- X NULL X NULL), (R1R2- X NULL), and R1R2R3 respectively.

The (GROUP BY . . . HAVING) operation in Query A has special semantics associated with it. For a given group of (r1.#, r2.#), if (r2.c OP2 COUNT(R3.*)) is true, the (r1.#, r2.#) group is passed along to Query B. However, if (r2.c OP2 COUNT(R3.*)) is false, the (r1.#, r2.#) group cannot be discarded. If the (r1.#, r2.#) is discarded and if this is the only group in which r1 was present, COUNT(R2.*) associated with the r1 tuple is 0 and hence should be preserved. If (r1.b OP1 0) is true, rl will be part of the result. The (r1, r2) tuple that does not satisfy (r2.c OP2 COUNT(R3.*)) should be passed along to Query B as (r1, NULL). Similarly, for tuples in the set (R1- X NULL X NULL), the GROUP BY . . . HAVING operation passes them as (R1- X NULL) to Query B because the predicate (r2.c OP2 COUNT(R3.*)) is false as (NULL OP 0) is false.

3.3.2 An Improved Dataflow Algorithm

In this subsection, we will discuss a new dataflow algorithm for nested queries with COUNTS (and hence outer joins). We will illustrate how we would have routed the three block query presented in Example 1, using the conventional dataflow algorithm. The execution tree for the query in Example 1 is shown in Figure 3.1. The nodes in the execution tree represent operations (restriction/join/group by etc.), while the directed arcs represent information flow. Tuples always flow in an upward direction. In Figure 3.1, it is important to notice that the conventional dataflow algorithm sends the joining tuples as well as the anti-join tuples to the immediate higher (parent) node. The sets that are propagated between the operators of the execution tree of Figure 3.1 are shown along the respective edges (using the notation of Section 3.3.1).

The set R1R2' is derived from those tuples in R1R2R3 and (R1R2- X NULL) that satisfy the predicate (R2.c OP2 COUNT(R3.*)), while the set (R1' X NULL) is derived from those tuples in R1R2R3 and (R1R2- X NULL) that don't satisfy the above predicate. Notice that the attributes of R2 have been replaced by NULL for these tuples.

In the presence of outer joins, a better execution tree may be obtained by sending the anti-join tuples to a node possibly higher than the parent node in the execution tree. This will result in savings in message and processing (CPU) costs. The new execution tree is shown in Figure 3.2.

In the first execution tree (Figure 3.1), we are shipping the (R1- X NULL X NULL) tuples (from the second outer join node to the first group by node) and

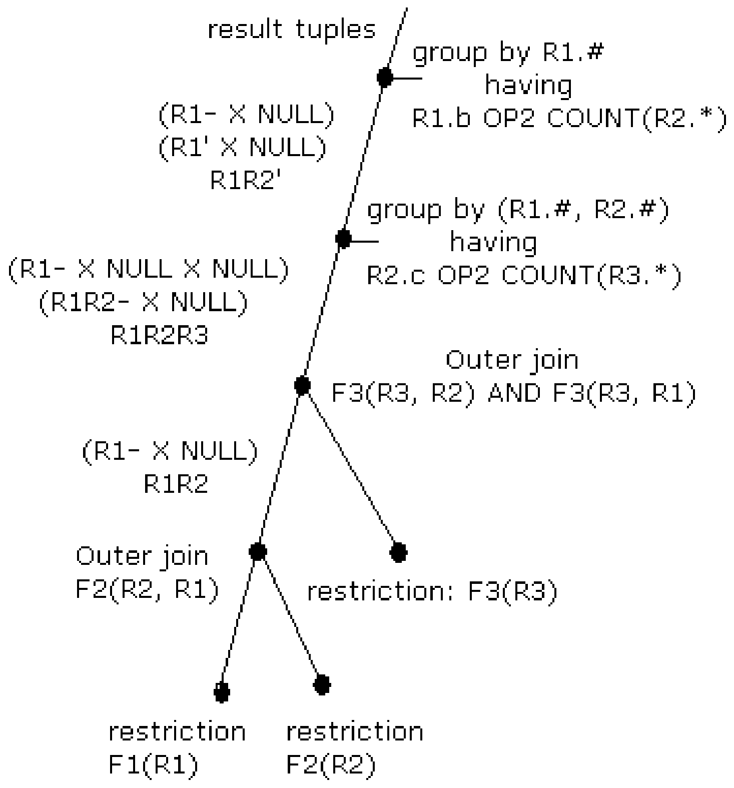

Figure 3.1: Conventional Routing Method

the (R1- X NULL) tuples (from the first group by node to the second group by node) unnecessarily. By doing so, we also incur the cost of processing them. In the second execution tree (Figure 3.2), the (R1- X NULL) tuples are shipped directly from the first outer join node to the second group by node.

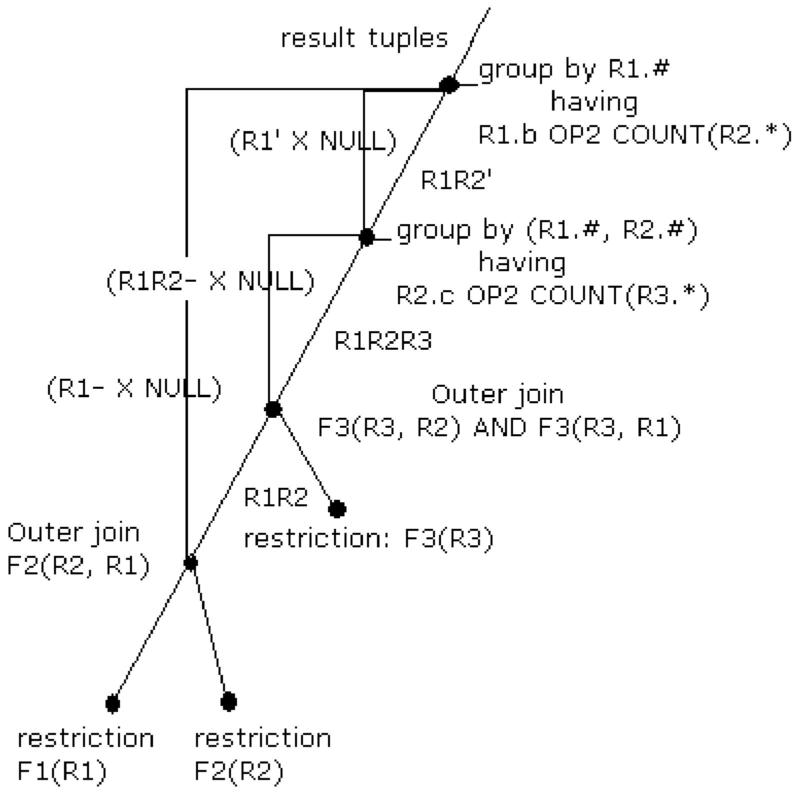

Figure 3.2: Improved Routing Method

Chapter 4

Modified Query plans

In this Chapter, we describe the modifications to the query plans for unnesting nested queries which are generated by the Integrated algorithm proposed by M. Muralikrishna. We first study the Kim's modified algorithm to avoid the COUNT bug. This modified Kim's approach is more efficient than Ganski's solution. Now we study queries with two blocks and then study queries with three or more blocks.

4.1 Queries with two blocks

Now we take the following example of a 2 block JA type nested query:

```
SELECT R.a
FROM   R
WHERE  R.b OP1 (SELECT COUNT(S.*)
                FROM   S
                WHERE  R.c = S.c)
```

The temporary table created in Kim's original unnesting algorithm remains unchanged. However, base query has to be modified. We know that the COUNT associated with a tuple of R that does not join with any tuple of S is 0. Thus, a tuple of r ∈ R that does not join with any tuple of TEMP1 will be a result tuple if (r.b OP1 0) is true. For a tuple r ∈R that joins with a tuple of TEMPI, r will be a result tuple if (r.b OP1 TEMP1.count) is true. The join operator in Query 2 is replaced by an outer join. In addition, different predicates are applied to the join (matching) tuples and the anti-join (non-matching) tuples to determine if they belong to the result. Notationally, we write this as shown below:

Query 1:

```
TEMP1(c,count) =    SELECT     S.c, COUNT(S.*)
                    FROM       S
                    GROUP BY   S.c
```

Query 2:

```
SELECT R.a
FROM   R, TEMP1
```

```
WHERE  R.c = TEMP1.c - - - - - - OJ
    [R.b OP1 TEMP1.count : R.b OP1 0]
```

The square brackets, in the last line of the above query, enclose the two predicates which are separated by a colon. The first predicate is applied to the joining tuples while the second tuple is applied to the anti-join tuples. Here we have proposed a successful implementation for the above query modifying the kim's algorithm. The Query 2 can be written in SQL as follows:

```
SELECT R.a
FROM   R LEFT OUTER JOIN TEMP1 ON R.c=TEMP1.c
WHERE  R.b = TEMP1.count OR
       (R1.b = 0 AND TEMP1.count IS NULL)
```

In the above query left outer join is taken as default outer join because we want to retain the anti join tuples of table R.

We now show that under certain circumstances, the modified Kim's method may be more efficient than Ganski's method. The heuristic argument is based on (1) the number of tuples that flow from each node in the query plans corresponding to the two methods and (2) the number of tuples that have to be processed at each group by and outer join node. The query plans for the two methods are shown in Figures 4.1. The edges in Figures 4.1 are labeled by the number of tuples flowing through those edges. Both methods involve accessing relations R and S. Clearly $|TEMP1| \leq |S|$ and $|R| \leq |R$ OJ $S|$. Assume that $|S| < |R|$. The number of tuples flowing from the group by node to the outer join node in Kim's method is equal to $|TEMP1|$. The number of tuples flowing from the outer join node to the group by node in Ganski's method is equal to $|R$ OJ $S|$. Clearly $|TEMP1| <|R$ OJ $S|$. The number of tuples processed by the groupby node and the outer join node in Kim's method is each less than the corresponding number of tuples in Ganski's method. Hence if $|S| < |R|$, Kim's method should perform better than Ganski's method. Here we need to notice one more fact that Ganski's method joins two base relations, whereas in Kim's method, we join a base relation with a temporary relation. As a result, Ganski's

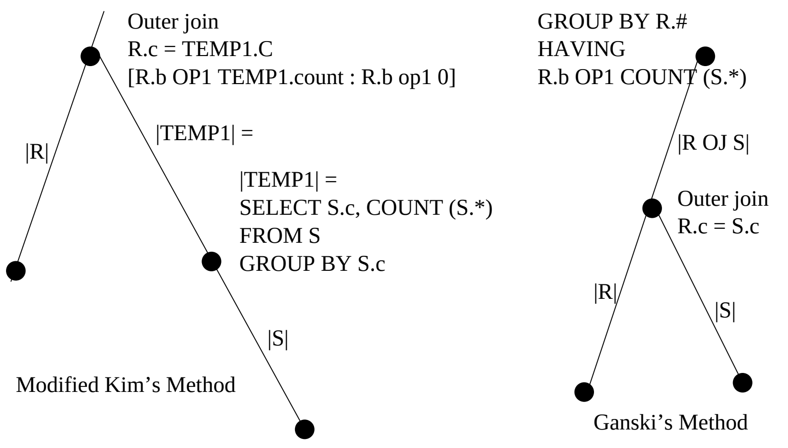

Figure 4.1: Modified Kim's Method Vs Ganski's Method

method might be able to employ more join methods. Clearly, the optimizer has to pick the cheaper method more carefully than as outlined above. The important point is that we can use Kim's method even in the presence of the COUNT aggregate when the correlation predicates are all equi joins.

4.2 Queries with three blocks

We now extend the modified Kim's algorithm to queries with three blocks. We first introduce a definition here.

Definition: An equi-join correlation predicate is called a neighbor predicate if it references the relation in its own block and the relation from the immediately enclosing block.

Consider the following example in which all the join predicates are neighbor predicates.

Example 2:

```
SELECT R.a
FROM   R
```

```
WHERE  R.b OP1 (SELECT COUNT(S.*)
                FROM   S
                WHERE  R.c = S.c
                AND    S.d OP2 (SELECT COUNT(T.*)
                                FROM   T
                                WHERE  S.e = T.e))
```

The algorithm given in [8] worked bottom up. We follow the same approach here. The result of the query is obtained by evaluating the following three unnested queries.

Query 3:

```
TEMP1(e, count) =   SELECT     T.e, COUNT(T.*)
                    FROM       T
                    GROUP BY   T.e
```

Query 4:

```
TEMP2(c,count) = SELECT S.c, COUNT(S.*)
                 FROM   S, TEMP1
                 WHERE  S.e = TEMP1.e - - - - - OJ
                        [S.d OP2 TEMP1.count : S.d OP2 0]
                 GROUP BY S.c
```

Query 5:

```
SELECT R.a
FROM   R, TEMP2
WHERE  R.c = TEMP2.c - - - - - - OJ
    [R.b OP1 TEMP2.count : R.b OP1 0]
```

Thus, we were able to extend the same principle to a three block query of Example 2 and avoid the COUNT bug. It is easy to see how we can extend the above solution to a query with more than three blocks as long as the correlation predicates are neighbor predicates. The natural question then is: what happens when we have non neighbor predicates. We address this in the next section.

4.3 Queries with non neighbor predicates

We start with the query shown in Example 3. This query is obtained by adding the non neighbor predicate, R.f = T.f, in the third block of the query in Example 2. Surprisingly, the query becomes very hard to unnest in the presence of the COUNT aggregates.

Example 3:

```
SELECT R.a
FROM   R
WHERE  R.b OP1 (SELECT COUNT(S.*)
                FROM   S
                WHERE  R.c = S.c
                AND    S.d OP2 (SELECT COUNT(T.*)
                                FROM   T
                                WHERE  S.e = T.e
                                WHERE  R.f = T.f))
```

Evaluating bottom up, we would expect the three unnested queries to be as follows:

Query 6:

```
TEMP1(e, f, count) =   SELECT      T.e,T.f, COUNT(T.*)
                       FROM        T
                       GROUP BY    T.e, T.f
```

Query 7:

```
TEMP2(c, f, count) = SELECT S.c, TEMP1.f, COUNT(S.*)
                     FROM   S, TEMP1
                     WHERE  S.e = TEMP1.e - - - - - OJ
                        [S.d OP2 TEMP1.count : S.d OP2 0]
                     GROUP BY S.c, TEMP1.f
```

Query 8:

```
SELECT R.a
FROM   R, TEMP2
```

WHERE R.c = TEMP2.c AND R.f = TEMP2.f - - - - - - OJ
[R.b OP1 TEMP2.count : R.b OP1 0]

There are no surprises in Queries 6 and 8. In Query 8, each tuple of R joins with at most one tuple of TEMP2. However, Query 7, as shown above, is incorrect!. The objective of Query 7 is to compute COUNT (S.*) associated with every (c, f) pair. Assume then that relations S and T are populated as shown in Figures 4.1 and 4.2.

c	**d**	**e**
10	2	100
10	0	100
10	1	200
10	0	200
10	3	200
10	0	300

Table 4.1: Table S

e	**f**	**g**
100	1000	1
100	1000	2
200	1000	3
200	2000	4

Table 4.2: Table T

Notice that we are selecting attributes from both S and TEMP1 in Query 7. We are also grouping by attributes from both the relations. In case an S tuple does not join with any TEMP1 tuples, we cannot meaningfully evaluate the query. Let us try to understand what happens when an S tuple does not join with any tuple of TEMP1. It is clear from the query of Example 3 that if an S tuple does not join with any T tuple, then COUNT (T.*) is 0, irrespective of the value of R.f. Therefore, such an S tuple will contribute to COUNT (S.*) if (S.d OP, 0) is true.

There is another subtlety that we need to focus on. Assume that a tuple s $\in$ S joins with one or more TEMP1 tuples. Let (TEMP1.f) denote the set of f values in the joining TEMP1 tuples. We need to decide if s will contribute to COUNT (S.*). If a tuple r $\in$ R has as an f value that is in {TEMP1.f} , we know that COUNT (T.*) associated with this (r, s) pair will be greater than 0. Then s will contribute to COUNT (S.*) if (S.d OP2 TEMP1.count) is true. On the other hand, for any tuple r $\in$ R that has an f value that is not in {TEMP1.f}, the corresponding COUNT (T.*) will be 0. If (S.d OP2 0) is true, then s will contribute to COUNT (S.*).

Using these observations, we now describe what the outer join operator of Query 7 must accomplish using the following pseudo code:

```
1   if no tuple of TEMP1 satisfies (S.e = TEMP1.e)
2   then output (S.c, all)
3   else for each tuple of TEMP1
4         satisfying (s.e = TEMP1.e)
5        {
6         if (S.d OP2 TEMP1.count)
7         then output (S.c, TEMP1.f)
8         else if (S.d OP2 0)
9              then output {S.c,  ~{TEMP1.f})
10        }
```

The pseudo code focuses on one S tuple, s, at a time. The second component of the tuple in Line 2 is a set that denotes all possible values of f. In Line 7 we output a tuple of the form (S.c, TEMP1.f). Line 9 indicates that we are outputting one tuple (S.c, $\backsim${TEMP1.f}). The second component of the above tuple is a set of values and is equal to the complement of the values present in set {TEMP1.f}. It is clear that the outer join operator has become more complex now! The group by operator in Query 7 is also a lot more complicated. The group by operator can easily determine the group to which a tuple from Line 7 belongs to as both the c and f values are available. A tuple from Line 2 logically belongs to all groups that have the same c value as this tuple since its f value represents all possible values of f. A tuple from Line 9 belongs to all groups that have the same c value as this tuple but whose (the group's) f value does not belong to set {TEMP1.f}. Logically, the number of such groups is bounded by the size of the domain of f. Potentially, this size could be infinite!

We further illustrate the complexity of the outer join and the group by operations in Query 7 using the data stored in the relations S and T. We first use

Query 6 to compute TEMP1.

e	**f**	**count**
100	1000	2
200	1000	1
200	2000	1

Table 4.3: TEMP1

Assuming OP2 denotes equality, the outer join operator of Query 7 produces the following output.

c	**second component**	**comments**
10	1000	from the 1^{st} tuple of S
10	$\backsim\{1000\}$	from the 2^{nd} tuple of S
10	1000	from the 3^{rd} tuple of S
10	2000	from the 3^{rd} tuple of S
10	$\backsim\{1000,2000\}$	from the 4^{th} tuple of S
10	all	from the 6^{th} tuple of S

It should be easy to see that the first, third, and the last tuple in the above table belong to the (10, 1000) group. The second, the fourth, and the last tuple belong to the (10,2000) group. Similarly, the second, the fifth, and the last tuple belong to the (10, x) group where x is any value except 1000 or 2000. The group by operation of query 7 must take the output of the outer join operator and produce TEMP2 (an infinite relation!) as shown in Table 4.4.

None of the f_i's in the above table are equal to 1000 or 2000. Notice that in this example, for every c value we have generated all possible f values, and hence the predicate (R.f = TEMP2.f) in Query 8 will always be satisfied. However, this predicate helps us identify the correct matching tuple in TEMP2. We have not been able to develop an efficient implementation for the group by operator of Query 7. Perhaps, it might be easier to modify the outer join of Query 8. Until a reasonable implementation is possible, we cannot employ Kim's method when a

c	f	count
10	1000	3
10	1000	3
10	f_1	3
10	f_2	3
10	f_3	3
...	...	...

Table 4.4: TEMP2

non neighbor predicate (T.f = R.f in this case) is present inside a COUNT block. However, if the second COUNT in Example 3 is replaced by a non COUNT aggregate, Query 7 would only have to perform a simple join. As in Dayal's solution, an outer join is used only when a COUNT aggregate is present between the blocks.

In the next two subsections, we shall present a couple of strategies that will enable us to generate more plans. The goal we are working towards is a new unnesting algorithm that incorporates the ideas presented above.

4.3.1 Precomputing the last aggregate

As we mentioned in Section 3.3, a valid J/OJ ordering is obtained by performing all the joins first, followed by the outer joins from left to right. Sometimes, we can change this order as demonstrated by the next query.

```
SELECT R.a
FROM   R
WHERE  R.b OP1 (SELECT COUNT(S.*)
                FROM   S
                AND    S.d OP2 (SELECT MAX(T.d)
                                FROM   T
                                WHERE  R.f OP3 T.f))
```

The J/OJ expression for the above query is R OJ(S J T). Since there is no correlation predicate between the S and T blocks, we have to perform a cartesian product to compute (S J T). The outer join is then performed using the predicate (R.f OP3 T.f). However, for each (r, s) pair, where r ∈ R and s ∈ S, MAX (T.d) depends only on r. Hence, we can precompute MAX(T.d) associated with each tuple of R as follows:

```
TEMP1(#, a, b, max)= (SELECT R.#, R.b, MAX(T.d)
                      FROM     R, T
                      WHERE   R.f OP3 T.f - - - - OJ
                      GROUP BY  R.#)
```

Notice that | TEMP1 | = | R |. Essentially, TEMP1 has all the attributes of R required for further processing along with the MAX (T.d) associated with each tuple of R. We were able to compute MAX (T.d) in this fashion only because it occurred in the last block. Any aggregate that does not occur in the last block depends on the results of the blocks below it and hence cannot be evaluated before the blocks below it are evaluated. Also, notice that we performed an outer join between R and T even though we were computing MAX (T.d). This is because COUNT (S.*) indirectly depends on each tuple of R as R is referenced inside the third block which is nested within the second block. Hence we must preserve all tuples of R. For a tuple of R with no joining tuples in T, the MAX value is set to NULL. We can now rewrite the original query as follows:

```
SELECT TEMP1.a
FROM   TEMP1
WHERE  TEMP1.b OP1(SELECT COUNT(S.*)
                   FROM    S
                   WHERE S.d OP2 TEMP1.max)
```

It is clear that it is possible to precompute the bottom most aggregate (BMA) if the number of outer relations referenced in the last block have already been joined. In the above example, the BMA depended only on one outer relation.

4.3.2 Performing outer joins before joins

As pointed out repeatedly, one correct evaluation order of a J/OJ expression is to perform the joins first followed by the outer joins from top to bottom. In this section we show that we may also proceed in a strictly top-down order, performing the joins and outer joins in the order they occur. As we shall see in Section 4.6, proceeding in a top down manner may enable us to use Kim's algorithm for a larger number of contiguous blocks at the end of the query. However, care must be taken to ensure that any join that is present just below an outer join is also evaluated as an outer join. We again illustrate with an example.

```
SELECT R.a
FROM   R
WHERE  R.b OP1 (SELECT COUNT(S.*)
                FROM   S
                WHERE  R.c OP2 S.c
                AND    S.d OP2 (SELECT MAX(T.e)
                                FROM   T
                                WHERE  S.e OP4 T.e
                                       AND R.f OP5 T.f))
```

The J/OJ expression is R OJ (S J T). The join predicate between S and T is (S.e OP4 T.e) and the outer join predicate is (R.c OP2 S.c AND R.f OP5 T.f). Assume that the join between S and T is very expensive and should be possibly avoided. Could we evaluate (R OJ S) first? It turns out that we can indeed perform (R OJ S) first. However, some precautions/modifications are necessary.

It is clear that if an R tuple has no matching S tuples, the count associated with that R tuple is 0. As pointed out in [10], this R tuple may be optionally routed to a higher node in the query tree so that it does not participate in the next join operation with T. We thus need to consider only the join tuples of the form (r, s) from the outer join, where r $\in$R and s $\in$ S. Let us focus our attention on a single tuple r of R. When the join with T is evaluated using the predicate (Se OP4 T.e AND R.f OP5 T.f), it is quite possible that none of these (r, s) tuples join with any tuples of T. In this case, the r tuple will be lost. However, if (r.b OP1 0) is true, r is a result tuple and hence must be preserved. On the other hand, if some of the (r, s) tuples do join with some T tuples, it may so happen that after we do the group by by (R.#, S.#) and evaluate MAX (T.d), none of the s.d values in the (r, s) tuples satisfy (s.d OP3 MAX (T.d)). We may be tempted to discard all the (r, s) groups. Again if (r.b OP1 0) is true, we need to preserve r.

We can preserve r if we perform the join between S and T as an outer join. Also, the group by operator must not discard any (r, s) group not satisfying (s.d OP3 MAX (T.d)). Instead, it must pass it on preserving the R portion of the tuple and nulling out the S portion of the tuple.

Similar ideas were used in [10] when unnesting tree queries. Summarizing, if we encounter the expression R OJ S OJ T J U J V, we could evaluate it as ((R OJ S) OJ (T J U J V)). The above order corresponds to evaluating all the joins first. Another evaluation order could be ((((R OJ S) OJ T) OJ (U J V))). Now we have an outer join between T and U. Carrying this idea one step further, the above expression may also be evaluated as ((((R OJ S) OJ T) OJ U) OJ V). As we shall see in the next section, joining relations in a top down order may enable us to employ Kim's method for a larger number of blocks.

4.4 An integrated algorithm

In this section we describe a new algorithm that generates execution plans by combining the ideas presented in above Sections. We have proposed the cheapest query plan among all others. Before we describe the new algorithm, we introduce a fairly simple graphical notation for JA type queries. The new algorithm will operate on graphs.

The graph G = (V, E) for a JA type query consists of a set of vertices V and a set of directed edges E. There is a one-one correspondence between the blocks of the query and the elements of V. Each element of V, except for the first vertex, is labeled either C (COUNT) or NC (Non COUNT). This labeling is clearly suggestive of the kind of aggregate (COUNT or Non COUNT) present in that block. The vertices are numbered 1 through d, where d is the current number of vertices in the graph. A directed edge is drawn from vertex i to j (i $<$ j) if there is a correlation predicate in the jth block between the relations of blocks i and j. In essence, the graph is a join graph.

Kim's method may be applied to the last k blocks of a query $(0 \leq k \leq d)$ if the last k vertices of the graph of the query satisfy the following properties:

- The in degree of every C vertex is at most 1.
- The edge incident on a C vertex corresponds to a neighbor predicate.
- All the edges incident with the last k vertices correspond to equi-join correlation predicates.
- The relations in the first d-k blocks have already been joined.

The BMA may be precomputed if the in degree of the last vertex is at most 1. The operations on the graph are as follows:

- When the relations of two or more blocks are joined, the corresponding vertices are collapsed into one vertex. The edges adjacent to these vertices are removed, while all the edges that connect these vertices to other vertices are preserved. Multiple edges are replaced by a single edge.
- Let d-l and d be the last two vertices in the graph. If the BMA is computed, the last vertex d is removed from the graph and the edge incident on d is connected to d-l.

Notice that we may be able to apply Kim's method only after joining some relations. For example, we may apply Kim's method to the last block after joining R and S in the query of Example 3. This is because the predicate (R.f = T.f) becomes a neighbor predicate only after relations R and S are joined. Thus, the number of blocks for which we may apply Kim's method can change dynamically. Similarly, the BMA may have originally depended on more than one outer relation but after these relations have been joined, the in degree of the last vertex will become 1. The BMA may be precomputed at this point.

When a series of consecutive m joins are encountered in a J/OJ expression, one may be tempted to evaluate all the joins using the cheapest order. It will become evident from the example at the end of this paper that we must evaluate joins incrementally. In other words, we must evaluate the first i joins at a time, where $1 \leq i \leq m$. This ensures that we may be able to apply Kim's method to a larger group of contiguous blocks at the end of the query.

We are finally ready to present the new algorithm, unnest, in pseudo code. The input to the algorithm is the graph G of the query and the output is a set of query plans. We shall not describe how the output is specifically constructed as this is implicit in the operations on the graph and should be fairly self evident. References to G' in unnest denote the new graph derived from G.

```
unnest(G)
{
  if (the BMA can be precomputed)
  { compute the aggregate.
    unnest (G');
  }
  if (Kims method can be applied to the
      remaining blocks)
     { apply Kims method
       return;
     }
  if (J--J-- ....---J---OJ---...) is encountered
     { for (i = 1; i <= m; i++)
       evaluate the first i joins using the
       cheapest join order.
       unnest (G');
     }
  if (OJ--J---J--.....---J---OJ---...) is encountered
     { for (i = 1; i <= m; i++)
       evaluate the first i joins using the
       cheapest join order.
       unnest (G');

       evaluate the first OJ; replace the
       first J by OJ.
       unnest (G');
```

```
        }
    }
```

We illustrate the working of the algorithm on the following query whose graph is shown in Figure 4.2.

```
SELECT R.a
FROM   R
WHERE  R.b OP1 (SELECT COUNT(S.*)
                FROM   S
                WHERE  R.c = S.c
                AND    S.d OP2 (SELECT AVG(T.e)
                                FROM   T
                                WHERE  S.e = T.e
                                       AND R.f = T.f
                                       AND T.g OP3 (SELECT SUM(U.g)
                                                    FROM   U
                                                    WHERE  S.h = U.h
                                                           AND T.i = U.i)))
```

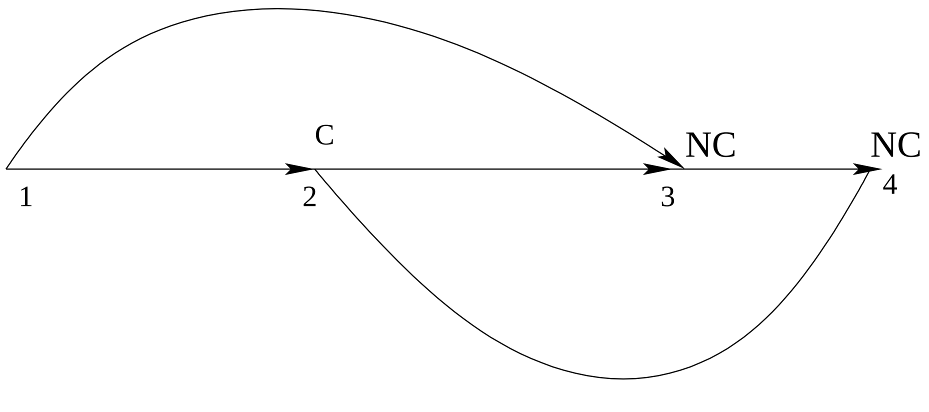

Figure 4.2: Graphical Representation of Integrated Algorihm

The J/OJ expression is R—OJ—S—J—T—J—U. The following query plans, as shown in Figures 4.3 to 4.8, are possible:

- (a) Apply Kim's method to blocks 2, 3, and 4.

- (b) Join R and S and apply Kim's method to blocks 3, and 4. Since the outer join between R and S is performed before the join, the first join is now evaluated as an outer join.
- (c) Join R, S, and T and apply Kim's method to block 4. Notice that both joins are now replaced by outer joins.
- (d) All joins have been replaced by outer joins, followed by three group by operations.
- (e) Join relations S, T, and U first, followed by the outer join. This amounts to applying the general solution for the entire query.
- (f) Join relations S, T, and U first. Since the BMA depends only on relations S and T, the BMA is computed before the outer join with R.

Notice that it was important to evaluate the joins incrementally. Most of the outer join nodes in Figures 4.3 to 4.8 have two output edges. The vertical edge represents the anti-join tuples, while the other edge represents the join tuples. Similarly, the groupby-having nodes have two output edges. The vertical edge represents the groups that did not satisfy that condition in the having clause. These groups have certain portions nulled out. For example, in Figure 4.6, groups flowing from the first groupby-having node to the topmost groupby-having node along the vertical edge are of the form (R, NULL) [10]. Also, Figures 4.3 to 4.8 have edges that route tuples to a node much higher in the tree than the immediate parent. As pointed out in [10], this optional but leads to savings in message costs.

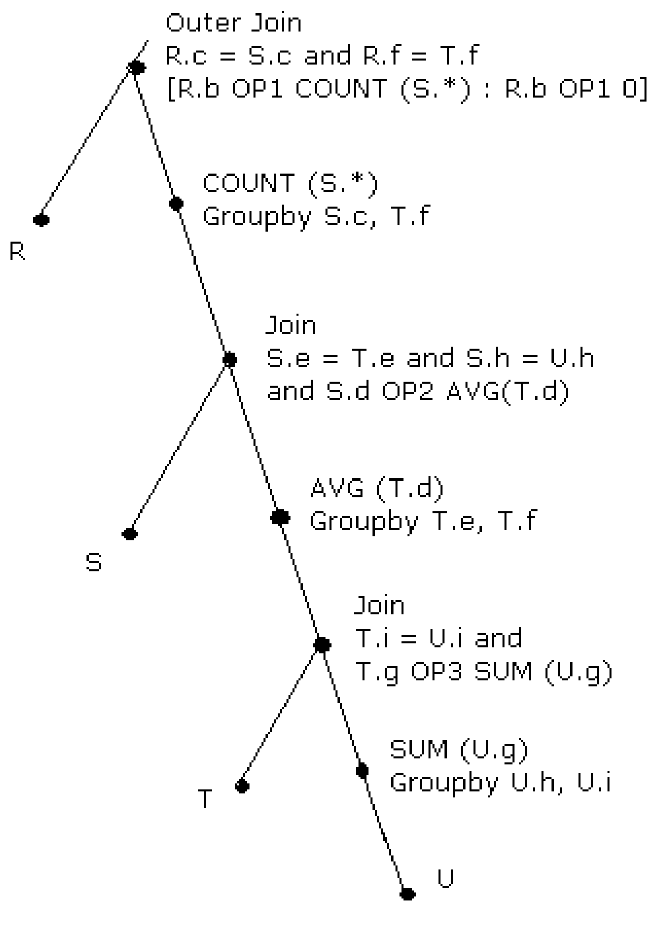

Figure 4.3: Query plan (a)

4.5 Flaws in Integrated Algorithm

The above explained Integrated algorithm contains some flaws in Query plans (b), (c) and (d). They are described as follows:

1. let us consider a case in Query plan (b) when certain join tuple from the outer join operation (R OJ S) doesn't find any match in outer join operation in the next higher layer and finding the path of anti join tuples to reach the

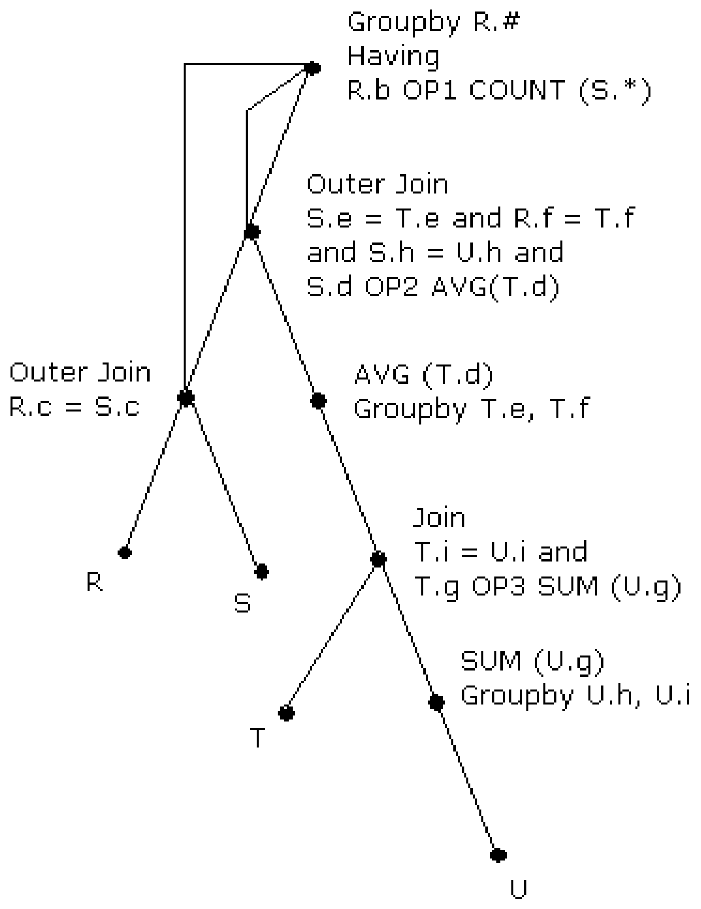

Figure 4.4: Query plan (b)

top most layer. In this case this anti join tuple is certain to come in the final outcome of the query if R.b is 0. But this not happening here because this anti tuple is carrying the primary key value of table S(Ex: S.srn here) to the top most layer. There the predicate (R.b OP1 COUNT(S.srn)) becomes false, since COUNT(S.srn) in not zero for the reason S.srn is not null.

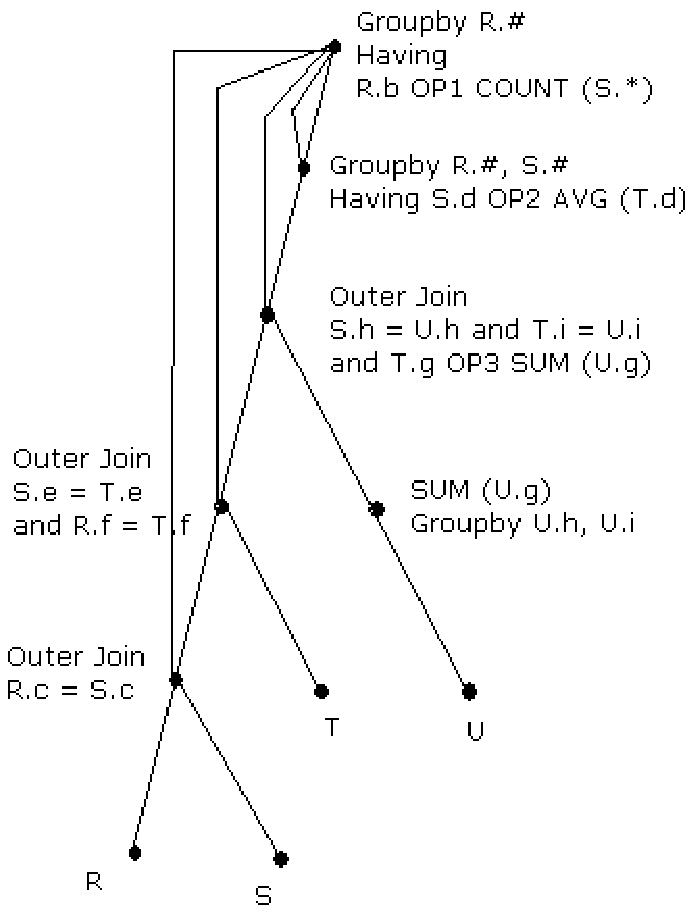

Figure 4.5: Query plan (c)

2. The same problem in Query plan (c) also, the outer join operation between the join tuples of outer join (R OJ S) and table T is propagating the primary key value of table S to the top most layer. There these anti join tuples are certain to appear in the final outcome of the query if R.b is 0. But this not happening because the predicate (R.b OP1 COUNT(S.srn)) becomes false, since COUNT(S.srn) in not zero for the reason S.srn is not null.

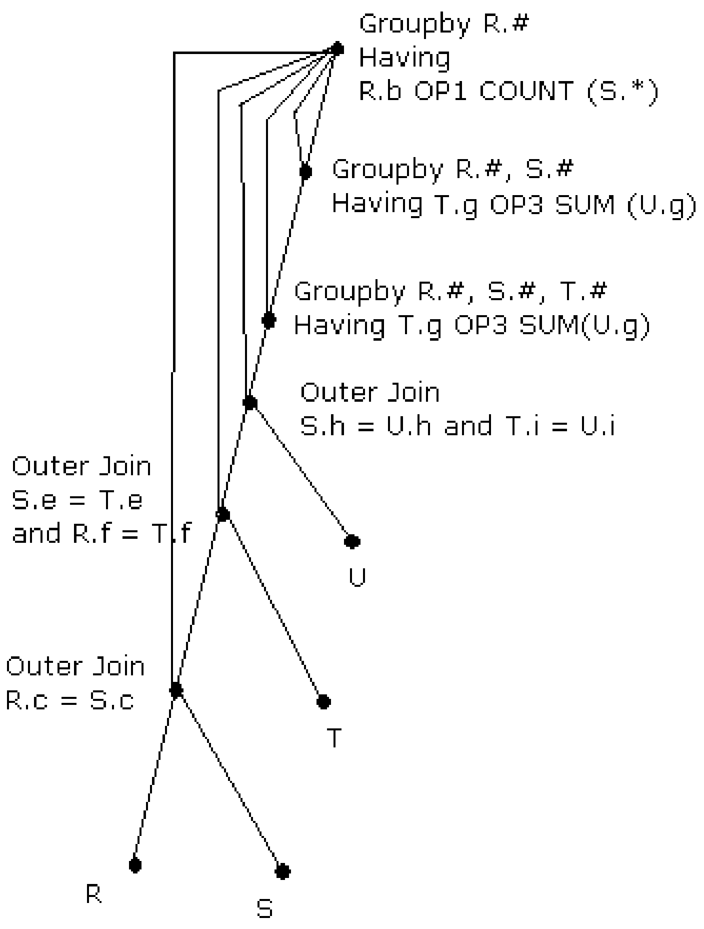

Figure 4.6: Query plan (d)

3. The same problem in Query plan (d) also, as explained above.

This is a common problem for all types of query plans which join R and S tables pairwise. The general problem lies when we join two tables which has a COUNT aggregate function between them. In the above example the problem has occurred in the query plans when we join tables R and S pairwise as there is

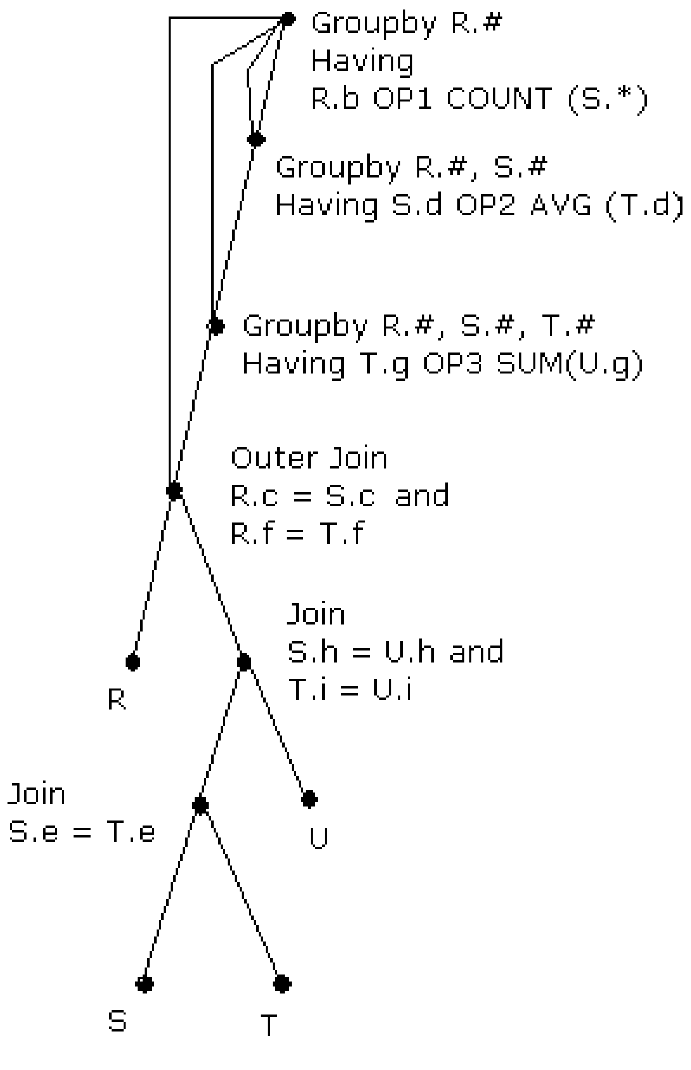

Figure 4.7: Query plan (e)

a COUNT aggregate function between them. The solution is given below.

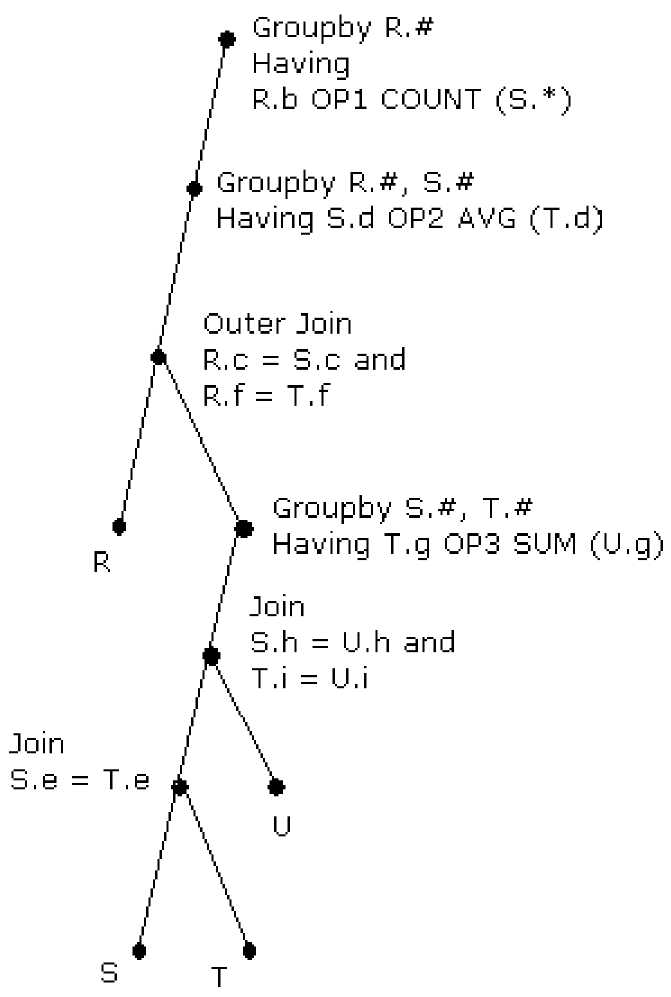

Figure 4.8: Query plan (f)

4.6 Solutions to the flaws in Integrated algorithm

Join the tables R, S and the resultant table of the right wing together. Then propagate the primary key value of 3rd table to the top most layer as it is 2nd table primary key value. This change is only for anti join tuples resulted from the outer join and where as for join tuples it would be same as it was before. In this case, S.srn would be null for all anti join tuples and then (R.b OP1 COUNT(S.srn)) becomes true if R.b is zero. Then this tuple appears in the final outcome of the query as the predicate (R.b OP1 COUNT(S.srn)) becomes true.

Applying the above solution to the Query plan(b) becomes joining(Outer join) the tables R, S and the resultant table of the right wing together and passing the primary key value of the 3rd table to the top most layer as it is primary key value of table S. And also same for Query plans (c) and (d), joining(Outer join) tables R, S and T together and propagate the primary key value of table T (Ex: T.srn here) to top most layer. Like this, the solution works fine for all Query plans. Now the Query plans (b), (c) and (d) are changed to as shown in Figure 4.9, 4.10 and 4.11 respectively.

The general solution is not to join the two tables when there exists a COUNT aggregate function between them, and to join these two tables with another table together in a query plan of unnesting a nested query.

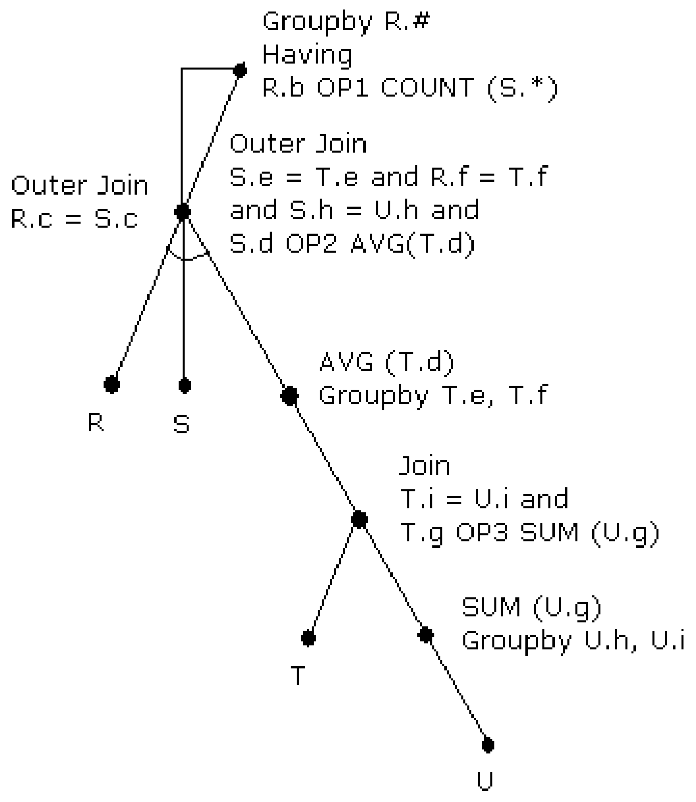

Figure 4.9: Modified Query plan (b)

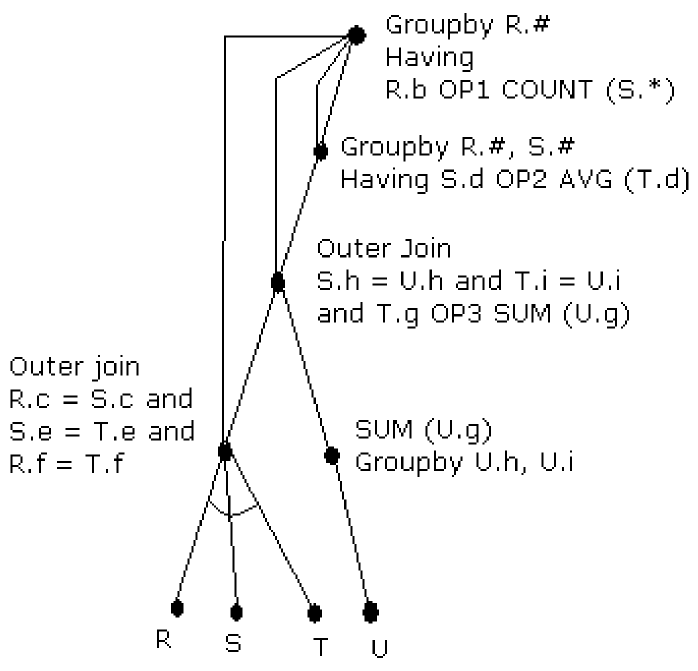

Figure 4.10: Modified Query plan (c)

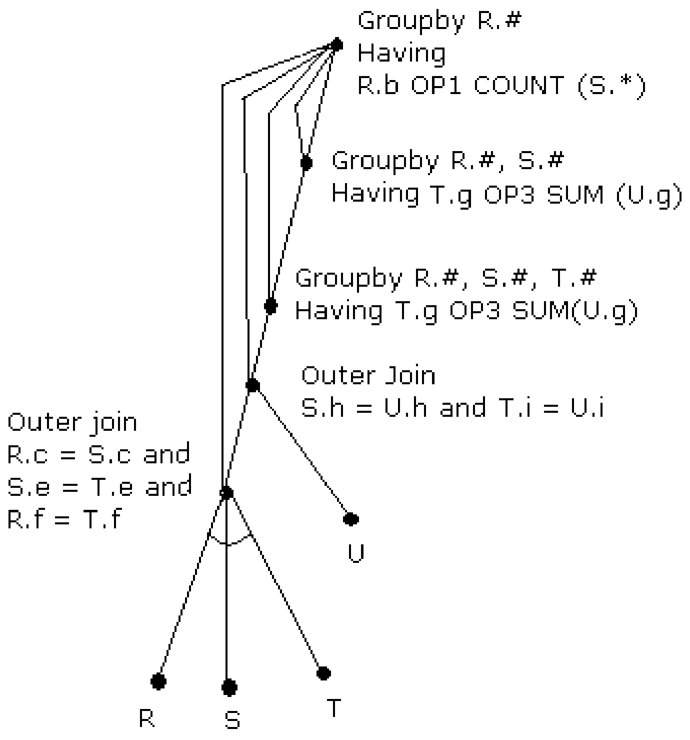

Figure 4.11: Modified Query plan (d)

Chapter 5

Implementation

As described in previous chapters, Kim's modified algorithm is an extension to original Kim's algorithm of unnesting the nested queries. This algorithm avoids the COUNT bug successfully in all cases of nested queries. And also this shows the better performance as compared to the later advancements so far after Kim's original unnesting algorithm. In terms of the execution time, Kim's modified algorithm is the computationally better unnesting algorithm over the techniques presented in [6], [4]. This performance analysis is described in the following section.

To implement the routing methods shown in Section 3.3.2 we used the temporary tables for the propagation of intermediate data to the upper layers query execution plan. The data set of tables R, S, T and U is used to implement the unnesting query plans for the given 4 block nested query in section 4.4. Here also we used temporary tables for the propagation of intermediate data to the upper layers in the implementation of query plans. We have carried out these experiments for different data sets of varying sizes from 100 to 1000 tuples in each relation. These results are taken as average of some possible iterative execution of each query plan. The flaws are identified in query plans generated by integrated algorithm and their solutions are also explained in the Sections 4.5 and 4.6. The performance analysis of all those query plans is described in following Section.

5.1 Performance analysis

To verify the efficiency of unnesting algorithm Kim's modified algorithm, we have taken 3 block nested query and generated the query plans using Nested iteration approach, Ganski's approach and Kim's modified algorithm. For each query, we measured the average execution time of multiple runs of the query as primary performance metric. The graphs of the results plot the elapsed time on Y-axis and the size of the data tables. The size of the tables denotes the number of tuples in each relation table in the query. Here we have taken nearly same no.of tuples in each table. We chose the size of the table as a parameter due to the fact that it directly relates to the intermediate result, which in turn, relates to the overhead corresponding to fetching tuples from the SQL engine. There by the elapsed time varies relative to the size of the table for all unnesting algorithms.

Implementations are done in DB2 server with the configuration, IBM eServer POWER5 520, 2-way 64 bit 1.50GHz., L2 cache 1.9MB, L3 cache 32Mb, 2 GB@ 266MHz. DDRI SDRAM, 2 X 73.4 GB HS U3 SCSI HDD with RAS features.

Our first experiment was done on 3 block nested query shown in example 2 of Section 4.2. The experimental results are shown in figure 5.1.

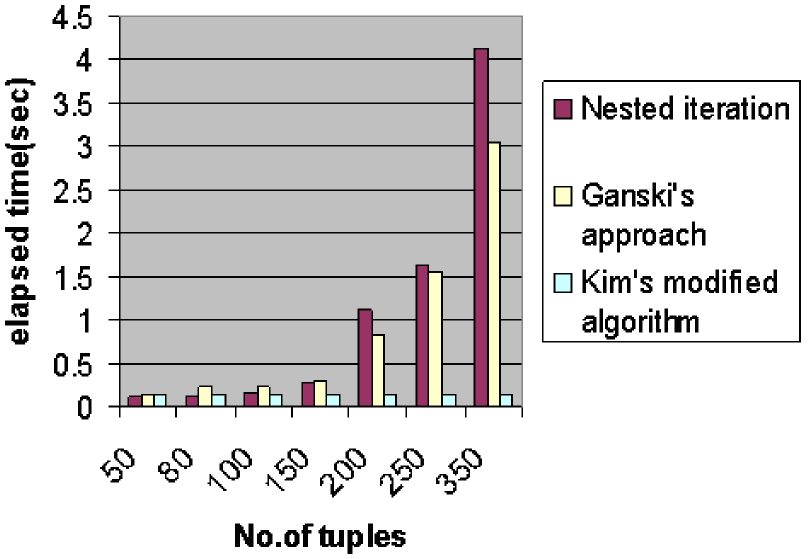

Figure 5.1: Performance show of Kim's modified algorithm

The next experiment was the performance analysis of all query plans generated by the integrated algorithm for the given 4 block nested query. The experimental results are shown in figures 5.2, 5.3 and 5.4. Here we have taken modified query plan (b), modified query plan (c) and modified query plan (d) as a replacement for query plans (b), (c) and (d) given in [9] respectively. This is due to the flaws they have, which we have already explained in Section 4.5.

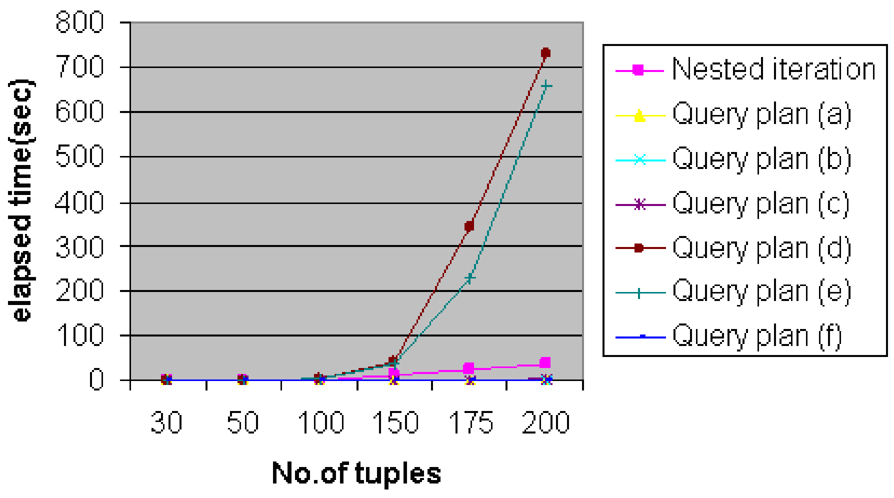

Figure 5.2: Nested iteration Vs unnesting query plans

Figure 5.2 shows the performance comparison of original nested iteration query with the unnesting the query plans for the same query. Query plans (a) and (b) shows better performance among all other query plans. Where as Figure 5.3 is only a comparison plot of Nested iteration query with query plans (a), (b), (c) and (d), because the other 2 query plans (d) and (f) are more expensive than the nested query in terms of elapsed time. Thats why for better exploitation of result we have omitted those 2 query plans from the results. And the same thing in Figure 5.4 we have shown only query plans (a), (b), (c) and (d). With the above experimental setup we can conclude that query plans (a) and (b) shows significantly grater reduction in execution time of unnested query.

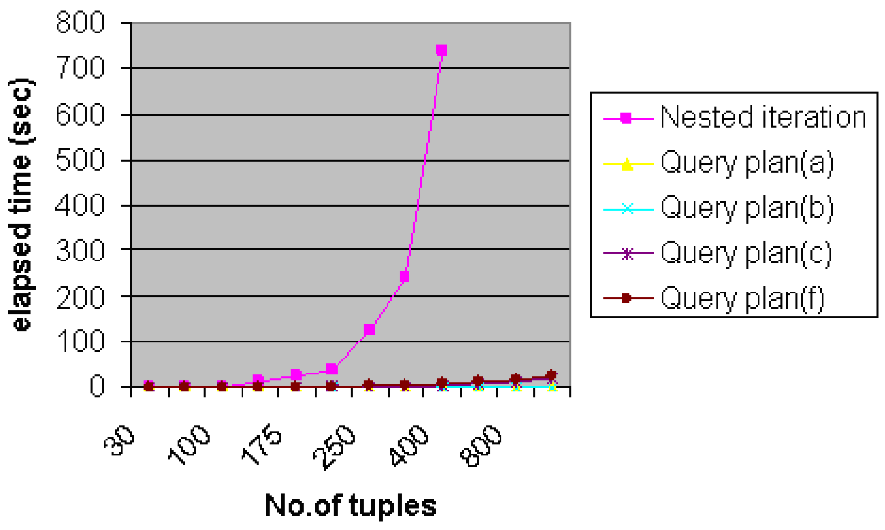

Figure 5.3: Nested iteration Vs query plans (a),(b),(c) and (f)

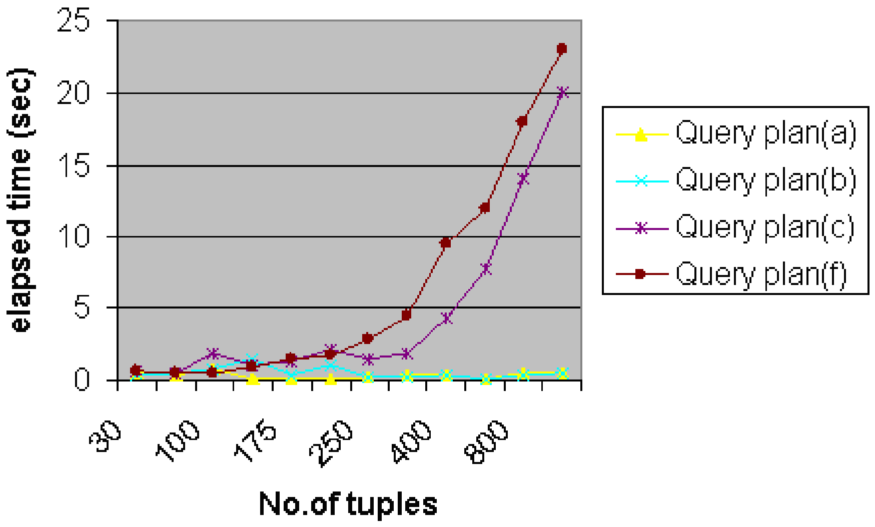

Figure 5.4: Performance show of query plans (a),(b),(c) and (f)

Chapter 6

Conclusions and Future work

6.1 Conclusions

In this thesis, we have given a successful implementation for a proposal made by M. Muralikrishna to avoid a COUNT bug in Kim's algorithm. In addition to this we have also improved the query plans generated by Integrated algorithm proposed by him, which enhances and incorporates the previously known techniques for unnesting JA type queries. This improvement is by eliminating the flaws in this algorithm. And also we have proved that one query plan (i.e Query plan (a)-applying Kim's method to all query blocks) among the all other as the cheapest one. There is one more query plan(i.e Query plan(b)-joining R and S and apply Kim's method to blocks 3, and 4) which is also giving very close results to this one, but the former one is the best.

6.2 Future work

As far as now we have not been successful in unnesting queries with non neighbor predicates, we are in the process of finding a meaningful implementation for this. It appears that we can unnest the various SQL predicates using the techniques presented in this thesis. It is hoped that more commercial systems will unnest SQL queries in the near future by creating an interface to the query optimizer of commercial database machines with the implementation of Integrated algorithm explained Chapter 4.

Bibliography

[1] Astrahan, M. M, and Chamberlin, "Implementation of a structured english query language," *ACM*, vol. 18, no. 10, pp. 580–588, October 1975.

[2] Astrahn, M. M, B. M. W, Chamberlin, D. D, Eswaran, K. P, Gray, J. N, G. P. P, K. W. F, Lone.RA, McJones, PR.Mehl, J. W, P. R, T. L, Wade, B. W, and Watscit, "System r relational approach to database management," *ACM Trans Database Syst*, vol. 1, no. 2, pp. 97–137, June 1976.

[3] Codd and E. F, "Extending the database relational model to capture more meaning," *ACM Trans Database Syst*, vol. 4, no. 4, pp. 397–434, December 1979.

[4] U. Dayal, "Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers," *Proc. VLDB Conf.*, pp. 197–202, September 1987.

[5] R. Epstein, "Techniques for processing of aggregates in relational database systems," *ERL/UCB Memo M79/8, Electronics Research Laboratory, Univ. of California, Berkeley*, February 1979.

[6] R. A. Ganski, "Optimization of nested sql queries revisited," *Proc. SIGMOD Conf.*, vol. 16, pp. 22–33, may 1987.

[7] W. Kiessling, "Sql-like and quel-like correlation queries with aggregates revisited," *UCB-ERL Memorandum No. 84/75. University of California, Berkley*, September 1984.

[8] W. Kim, "On optimizing an sql-like nested query," *ACM Transactions on Database Systems (TODS)*, vol. 7, no. 3, pp. 443 – 469, 1982.

[9] M.Muralikrishna, "Improved unnesting algorithms for join aggregate sql queries," *Proceedings of the 18th International Conference on Very Large Data Bases*, pp. 91 – 102, 1992.

[10] M. Muralikrishna, "Optimization and dataflow algorithms for nested tree queries," *Very Large Data Bases Proceedings of the 15th international conference on Very Large data bases*, pp. 77–89, August 1989.

[11] A. Rosenthal and C. Galindo-Legaria, "Query graphs, implementing tress, and freelyreorderable outerjoins," *Proc. SIGMOD Conf.*, pp. 291–299, May 1990.