SQL-Like and Quel-Like Correlation Queries with Aggregates Revisited

by

Werner Kiessling

Memorandum No. UCB/ERL 84/75

17 September 1984

(cover)

Electronics Research Laboratory

SQL-LIKE AND QUEL-LIKE CORRELATION QUERIES WITH AGGREGATES REVISITED

by

Werner Kiessling

Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA.

Abstract

Recent transformation algorithms for speeding up evaluation of nested SQL-like queries with aggregates are reviewed with respect to the correctness for aggregates over empty sets. It turns out that for a particular subset of such queries these algorithms fail to compute consistent answers. Unfortunately there seems to be no way that these transformations can be made correctly under all circumstances. Also the algorithms for QUEL are reexamined, likewise revealing a bug for nesting levels greater than one. However, in contrast, this error can easily be fixed. Moreover, as a valuable by-product from the user's view, for a specific subset of QUEL-queries with aggregates a simple semantics can be associated a posteriori.

This report was prepared while the author was on leave from the Technische Universitaet Muenchen, Institut fuer Informatik, West Germany.
Die Arbeit wurde mit Unterstuetzung eines Stipendiums des Wissenschaftsausschuss der NATO ueber den DAAD ermoeglicht.

1. INTRODUCTION

This note is concerned with a specific type of nested queries, involving aggregates and correlated predicates in a nested query block. This query type is exemplified by the following sample relations and SQL-query:

Relations:

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

Query Q1:

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

Each PARTS tuple contains the part number and the actual quantity on hand. In the SUPPLY relation each tuple has a part number and information on the quantity of that part shipped at a particular date. The meaning of query Q1 is:
Find the part numbers of parts whose quantites on hand equal the highest quantities of those parts shipped before 1-1-80.

A similar example can be found in [KIM82]. In his paper a terminology for nested SQL-like queries is developed. We will exactly be concerned with what is there termed *type-JA queries*. A nested query Q is of type JA if the WHERE clause of the inner block contains a join predicate that references the relation of the outer block (PARTS.PNUM in our example) and an aggregate function (here MAX) is associated with the SELECT clause of the inner block. Similar classifications of nested SQL-like queries can also be found in [MAK81] and [KIE83].
The standard method for evaluating correlation queries with aggregates is by *nested-iteration*, evaluating the inner query block once for each substituted correlation value (PARTS.PNUM-value in our case).[1] By this nested-iteration procedure also the semantics of a type-JA SQL-query is defined in a convenient way. As a matter of fact, for a large set of such queries and database characteristics this method suffers from very poor performance. In order to design faster algorithms obviously one has to look for ways where the inner query block, calculating the aggregates, may be evaluated only once. This has been recognized independently at several sites and has led to the design of alternate algorithms to process correlation queries with aggregates, see [KIM82] and [KIE83] for SQL-like queries and [EPS79] for QUEL-like queries. While in the first two works the nested-iteration semantics is taken as a starting point from which hopefully equivalent transformation algorithms can be developed, the last work defines the semantics of QUEL-queries using aggregates entirely by the given evaluation algorithm and no attempt is made to relate the semantics thereby defined to a more easily comprehensible semantics like nested-iteration. As an additional means to speed up the processing of correlation queries with aggregates the utilization of dynamic filters is desirable, as described in [KIE84].
However, these semantic transformations must be treated very carefully in order to give consistent results with respect to the original semantics, defined by nested-iteration. The main

[1] Ingres folks are reminded that in SQL there is an implicit binding of PARTS.PNUM in the inner query block with the outer PARTS.QOH.

thrust of this note is to exhibit the kind of possible mistakes and ,more importantly, to show special cases where these semantic transformations fail. As it will turn out, the solutions given in [KIM82] and [KIE83] fail for some queries applying the COUNT aggregate. Unfortunately those algorithms cannot easily be adjusted to produce the desired answers. On the other hand, the algorithms given in [EPS79] for processing analogous QUEL-queries yields the desired results unless an additional optimization procedure is applied, which fails for nesting levels greater than one.

2. SOURCES OF INCONSISTENCY FOR SQL

2.1. Handling of aggregates over empty sets.

Let us assume the following instantiation of our PARTS and SUPPLY relations:

PARTS	PNUM	QOH
	3	6
	10	1
	8	0

SUPPLY	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

What should the desired answer for our given query Q1 be in this particular case? Evaluating Q1 by nested-iteration proceeds here as follows: Fetch each PARTS tuple, extract its PNUM value and substitute it into the inner query block where it replaces PARTS.PNUM. Thereafter the inner block is evaluated and its result compared to QOH of that tuple in question.
Doing so, the question how to handle aggregates over empty sets arise immediately. Let us distinguish two cases:

(a) The default value for aggregates over empty sets is set to zero.[2]

Result:	PARTS.PNUM
	10
	8

(b) Agggregates like AVG, SUM, MIN, MAX are set to a special NULL-value for empty sets. Additionally, expressions like "QOH = NULL" evaluate to the unknown truth value (denoted by ? in [CHA76]). The truth value of an entire WHERE clause is computed using three-valued logic ([CHA76]), whereby tuples are considered to not satisfy the WHERE clause if the overall truth value is false or ?.

[2] This is done e.g. in INGRES ([STO80]).

Result: PARTS.PNUM
10

Now we perform the transformation of query Q1 due to the algorithms described in [KIM82].

- First transform the type-JA query into a type-J query by NEST-JA algorithm ([KIM82]):

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

SELECT PNUM
FROM PARTS
WHERE QOH = ( SELECT MAXQUAN
              FROM TEMP
              WHERE TEMP.SUPPNUM = PARTS.PNUM )
```

This transformation results in materializing a temporary relation TEMP and in a query of type J.

- Second transform the type-J query by NEST-N-J algorithm ([KIM82]):

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

This resulting query now is a vanilla join query.
In order to materialize the temporary relation TEMP we recall the semantics of SQL-queries with a WHERE clause and a GROUP BY clause ([CHA76]): First the WHERE clause is applied to qualify tuples, then the respective groups are formed and then an aggregate function is applied to each group.
This semantics produces:

TEMP	SUPPNUM	MAXQUAN
	3	4
	10	1

The evaluation of our query Q1 following this semantic transformation yields the following:

Result: PARTS.PNUM
10

As can be seen, this result matches that of employing NULL-values for the nested-iteration semantics.

2.2. Troubles with the COUNT-aggregate.

Well, up until now everything seems to work quite nicely, assuming the proper use of NULL-values which assures the equivalence of the transformation algorithm of [KIM82] to the nested-iteration semantics. (This remark also applies to the related algorithms in [KIE83].) There seem to be troubles with this semantic transformation only in the event of an inadequate treatment of aggregates over empty sets. Unfortunately there are examples where there is no way out of this dilemma of aggregates over empty sets and which prove that this type of semantic transformation for type-JA SQL-queries is incorrect for a certain subset of JA-type queries. To demonstrate this, we will evaluate a query Q2 which is derived from our original Q1 by simply substituting the aggregate MAX by COUNT. The reason for choosing COUNT instead of MAX is that COUNT is a totally defined function, i.e. COUNT over the empty set is defined as zero. (This implies also that for this new example the existence/non-existence of NULL-values is irrelevant.)

Query Q2:

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

The meaning of query Q2 is supposed to be as follows:
Find the part numbers of those parts whose quantities on hand equal the number of shipments of those parts before 1-1-80.

Evaluating Q2 due to nested-iteration semantics yields:

```
Result:  PARTS.PNUM
           10
           8
```

The transformation of Q2 using Kim's algorithms gives the following outcome:

```
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
```

Evaluation of the above yields (remember the semantics of WHERE...GROUP BY...):

TEMP'	SUPPNUM	CT
	3	2
	10	1

Result:	PARTS.PNUM
	10

Again the results differ. But unfortunately we now have no resort to establish a match because COUNT is a totally defined function. The very reason, why this transformation fails in this particular case is as follows: due to the WHERE...GROUP BY semantics, materializing TEMP' eliminates non-present SUPPLY.PNUM values; thus these empty sets are not counted and evaluated to CT = 0. On the other hand, the nested-iteration method accounts for counting empty sets. As a matter of fact, similar deficiencies will show up for the case where the outer correlation column (PARTS.PNUM) is not a subset of the inner correlation column (SUPPL.PNUM), as in our example.
Remark: The loophole in Kim's paper lies in the proof of his lemma2 upon which his NEST-JA algorithm relies. An existential quantifier is implicitly assumed when it is stated: "*Then it is clear that the query may be processed by fetching each tuple of Ri, then fetching the Rt tuple whose Cl column has the same value as the Cp column of the Ri tupie....*"
(In our example the roles of Ri, Rt, C1 and Cp are occupied by PARTS, TEMP', PARTS.PNUM and TEMP'.SUPPNUM.)

How to fix these bugs?

If one does not want to resort to a quite different algorithm then the following modification comes into mind:
Trial correction: Adjust the transformation algorithm for JA-queries involving COUNTs by a posteriori recovering lost aggregates over empty sets in the following way:
TEMP' is defined as before, however the remaining query is modified into

```
SELECT PNUM
FROM PARTS,TEMP'
WHERE ( PARTS.QOH = TEMP'.CT AND
        PARTS.PNUM = TEMP'.SUPPNUM )
      OR
      ( PARTS.QOH = 0 AND
        PARTS.PNUM IS NOT IN ( SELECT SUPPNUM
                               FROM TEMP' ) )
```

Unfortunately this solution only works for 1-level deep correlated type-JA queries like our Q2. The following, somewhat more complex, example shows that in general the transformation under consideration cannot be fixed for arbitrarily correlated SQL-queries using COUNTs.

Relations:

Ri Ck	Ch	Cr
3	6	3
10	1	10
8	0	8
5	1	5
15	0	15
12	0	12

Rj Cm	Cn
79	4
84	2
78	1
81	1
77	5
82	6
74	0
83	12

Rk Cp	Cq
1	8
2	8
3	8
3	5
4	3
5	10
1	15
2	15
4	15

Query Q3:

```
SELECT Ri.Ck
FROM   Ri
WHERE  Ri.Ch = ( SELECT COUNT( Rj.Cm )
                 FROM Rj
                 WHERE Rj.Cn = ( SELECT COUNT( Rk.Cp )
                                 FROM Rk
                                 WHERE Rk.Cq = Ri.Cr
                                 AND   Rk.Cp != 3 )
                 AND   Rj.Cm < 80 )
```

Expected result due to nested-iteration semantics: 10, 8, 5, 15

Following the transformation of [KIM82], Q3 would be processed as follows:

```
Rt1(C1, C2) = ( SELECT Rk.Cq, COUNT( Rk.Cp )
                FROM Rk
                WHERE Rk.Cp != 3
                GROUP BY Rk.Cq )

Rt2(C1, C2) = ( SELECT Rt1.C1, COUNT( Rj.Cm )
                FROM Rt1, Rj
                WHERE Rj.Cn = Rt1.C2 AND Rj.Cm < 80
                GROUP BY Rt1.C1 )

SELECT Ri.Ck
FROM   Ri, Rt2
WHERE  Ri.Ch = Rt2.C2 AND Ri.Cr = Rt2.C1
```

The two temporaries Rt1 and Rt2 materialize as follows:

Rt1 C1	C2
8	3
3	1
10	1

Rt2 C1	C2
3	1
10	1

Result for this original transformation: [10]

Can this mismatch be corrected afterwards by a modification like that for the 1-level nested query Q2? Modifying the final subquery above to

```
SELECT Ri.Ck
FROM Ri, Rt2
WHERE ( Ri.Ch = Rt2.C2 AND Ri.Cr = Rt2.C2 )
        OR
        ( Ri.Ch = 0 AND Ri.Cr IS NOT IN ( SELECT Rt2 FROM C2 ))
```

produces as result 10, 8, 15, thus recovering 8 and 15 lost by the outer COUNT. But unfortunately there seems to be no general way to recover values lost by COUNTs on a correlation level greater than 1. In this example recovering 5 cannot be done using the same approach as for the outer COUNT.
Consequently in order to devise a transformation which will produce consistent results for any type-JA query, different algorithms must be considered. As a preparation for the QUEL approach let us review the reason that the considered solution fails in some cases.
The transformation algorithm projects the inner correlating column (SUPPLY.PNUM for Q2) into a temporary (TEMP) for a subsequent GROUP BY computation. This is correct as long as the outer correlation column's values (PARTS.PNUM) are a subset of the inner ones. Otherwise some outer correlating values may not be accounted for in the final join.
Therefore a consistent algorithm should start by projecting the *outer* correlation column into a temporary relation, which is done by QUEL.

3. REVISITING QUEL SEMANTICS FOR AGGREGATES

3.1. Discussion of the QUEL Approach.

As mentioned introductorily the semantics of QUEL queries with aggregates are not defined that conveniently by nested-iteration. The reason is probably founded in the following complications with QUEL, as compared to SQL. Unlike SQL, QUEL distinguishes between simple aggregates and aggregate functions ([EPS79]). The difference is best shown by the following example, which at first glance could be thought to be the QUEL-equivalent of our SQL-query Q2:

```
RANGE OF P IS PARTS
RANGE OF S IS SUPPLY
RETRIEVE ( P.PNUM )
WHERE P.QOH = COUNT ( S.SHIPDATE
                    WHERE S.PNUM = P.PNUM AND S.SHIPDATE < 1-1-80 )
```

However, in this query P.PNUM is completely local to the aggregate COUNT, i.e. there is no linking between the outer P.QOH and the inner P.PNUM. This usage of an aggregate is termed a scalar aggregate, and it evaluates to a single value which is substituted to compute the outer query. If we want to write the QUEL equivalent to Q2, then we must explicitly establish this desired link by using a *BY-list* containing the outer correlation column P.PARTS for the COUNT aggregate, which is done below:

Query 2':

```
RANGE OF P IS PARTS
RANGE OF S IS SUPPLY
RETRIEVE ( P.PNUM )
WHERE P.QOH = COUNT ( S.SHIPDATE BY P.PNUM
                    WHERE S.PNUM = P.PNUM AND S.SHIPDATE < 1-1-80 )
```

In contrast to the type-JA SQL-queries where the user can conveniently think of the associated semantics in terms of nested-iteration, QUEL is unable to assign an analogously simple semantics to all correlation queries with aggregates. This is founded in the fact that due to the freedom of explicit bindings through BY-lists a much broader class of aggregate queries than in SQL is defined.[3] In fact, the semantics of QUEL-queries with aggregates is soly defined by the evaluation procedure described informally in [EPS79].[4] His algorithm is similar to those of [KIM82] and [KIE83] in so far that it likewise attempts to evaluate the inner block only once, but it avoids the pitfalls with the COUNT aggregate by initially projecting the outer correlation column into a temporary relation, which guarantees not to lose any outer correlation values. Applied to Q2', in principal this algorithm works as follows:

```
/* (1) Project outer correlation column (being exactly the BY-list) and initialize aggregates. */

RETRIEVE INTO TEMP1 ( P.PNUM , CT = 0 )

/* (2) Evaluate partial query with aggregate function locally, maintaining
the connection between the inner correlation column values and their
respective aggregate value. */
/* Be carefully not to remove duplicates for TEMP2a. */

RETRIEVE INTO TEMP2a ( P.PNUM , S.SHIPDATE )
WHERE S.PNUM = P.PNUM AND S.SHIPDATE < 1-1-80

RANGE OF T2a IS TEMP2a
RETRIEVE INTO TEMP2b ( T2a.PNUM, CT = COUNT( T2a.SHIPDATE BY T2a.PNUM )

/* (3) Replace aggregates over non-empty sets by their real values in TEMP1. */

RANGE OF T1 IS TEMP1
RANGE OF T2b IS TEMP2b
REPLACE T1 ( CT = T2b.CT ) WHERE T1.PNUM = T2b.PNUM

/* (4) Establish the link on PNUM and evaluate outer block. */

RETRIEVE ( P.PNUM )
WHERE P.QOH = T1.CT AND P.PNUM = T1.PNUM
```

The evaluation of Q2' due to this algorithm proceeds as follows:

/* Steps 1 - 3 */

TEMP1	PNUM	CT	TEMP2a	PNUM	SHIPDATE
	3	0		3	7-3-79
	10	0		3	10-1-78
	8	0		10	6-8-78

[3] However, it is strongly doubted whether this degree of freedom has any advantages over the restrictive implicit binding mechanism in SQL.

[4] This algorithm is implemented in University Ingres as well as in the commercial version RTI-Ingres.

TEMP2b	PNUM	CT	TEMP1	PNUM	CT
	3	2		3	2
	10	1		10	1
				8	0

Result: 10, 8.

Remarks:
(1) If we assume that there are no duplicate values for the outer correlation column PARTS.PNUM, then the correctness of this algorithm should be clear after all preceeding discussions (this proposition has to be taken modulus the remarks stated subsequently in section 3.2). In particular the reader should convince himself that this algorithm is not sensitive to a specific choice of the defaults for aggregates over empty sets. (Nevertheless, NULL values should be supported.) Note also that in general the join clause in step2 must not be dropped. If however for whatever reasons there are duplicate values in the outer correlation column then this algorithm fails to be equivalent to nested iteration. (In the contrary, this issue does not arise for KIM's algorithm.) In order to establish an equivalence to nested-iteration in every conceivable case the computation of TEMP2a would have to be changed as follows:

```
RANGE OF T1 IS TEMP1
RETRIEVE INTO TEMP2a ( T1.PNUM, S.SHIPDATE )
WHERE S.PNUM = T1.PNUM AND S.SHIPDATE < 1-1-80
```

(2) For type-JA queries involving aggregates different from COUNT this algorithm is slower compared to Kim's. However, it is capable of processing a larger class of correlation queries which are no longer of type-JA, e.g. consider the query

```
SELECT r1, r2 FROM R
WHERE r3 = ( SELECT AVG( s1 ) FROM S, T
             WHERE s2 = t1 AND r4 = t2 )
```

This query may be correctly processed using this algorithm[5], while [KIM82] is not directly applicable to it.

The most important result however is that we are now able to assign an easy semantics to a certain subclass of QUEL-queries with aggregates. Namely, if we consider the class of all QUEL-queries which is retained by translating type-JA SQL-queries into their QUEL-counterparts with the proper BY-list choice, then the algorithm of [EPS79] implements the nest-iteration semantics (up to the mentioned exceptions).

3.2 BY-list optimization in QUEL.

As stated, the result obtained concerning the correctness of [EPS79] holds only if a procedure called *BY-list optimization* in [EPS79] is not applied unconditionally. We will demonstrate the intension and the effect of this BY-list optimization by the QUEL-counterpart of our previous SQL-query Q3. The QUEL-equivalent of Q3 looks as follows:

[5] Also [KIE83] can process this larger class of correlation queries.

QUERY Q3':

```
RANGE OF I IS Ri
RANGE OF J IS Rj
RANGE OF K IS Rk
RETRIEVE ( I.Ck )
WHERE I.Ch = COUNT( J.Cm BY I.Cr
                    WHERE J.Cn = COUNT( K.Cp BY I.Cr
                                        WHERE K.Cq = I.Cr
                                        AND   K.Cp != 3 )
                    AND  J.Cm < 80 )
```

If we omit the BY-list optimization then Q3' is evaluated as follows (using the procedure given in the previous section 3.1):

Phase A: Evaluate innermost aggregate

(1) RETRIEVE INTO TEMP1 (I.Cr, CT1 = 0)

(2a) /* don't eliminate duplicates */
RETRIEVE INTO TEMP2a (I.Cr, K.Cp)
WHERE K.Cq = I.Cr AND K.Cp != 3

(2b) RANGE OF T2a IS TEMP2a
RETRIEVE INTO TEMP2B (T2a.Cr, CT1 = COUNT(T2a.Cp by T2a.Cr))

(3) RANGE OF T1 IS TEMP1
RANGE OF T2b IS TEMP2b
REPLACE T1 (CT1 = T2b.CT1) WHERE T1.Cr = T2b.Cr

Phase B: Replace innermost aggregate by computed result, accomplishing the proper links required by the BY-list.

/* The remaining query to be processed now is: */

```
RETRIEVE I.Ck
WHERE I.Ch = COUNT( J.Cm BY I.Cr
                    WHERE J.Cn = T1.CT AND J.Cm < 80
                    AND I.Cr = T1.Cr )
```

Phase C: Evaluate aggregate in above modified query.

(1') RETRIEVE INTO TEMP3 (I.Cr, CT2 = 0)

(2a') /* don't eliminate duplicates */
RETRIEVE INTO TEMP4a (I.Cr, J.Cm)
WHERE J.Cn = T1.CT1 AND J.Cm < 80
AND I.Cr = T1.Cr

(2b') RANGE OF T4a IS TEMP4a
RETRIEVE INTO TEMP4b (T4a.Cr, CT2 = COUNT(T4a.Cm BY T4a.Cr))

(3') RANGE OF T3 IS TEMP3
RANGE OF T4b IS TEMP4b
REPLACE T3 (CT2 = T4b.CT2) WHERE T3.Cr = T4b.Cr

Phase D: Process remaining query, being modified accordingly.

(4) RETRIEVE (I.Ck) WHERE I.Ch = T3.CT2 AND I.Cr = T3.Cr

/* Steps 1 - 3 */

TEMP1 Cr	CT1	TEMP2a Cr	Cp	TEMP2b Cr	CT1
3	0	3	4	3	1
10	0	10	5	10	1
8	0	8	1	8	2
5	0	8	2	15	3
15	0	15	1		
12	0	15	2		
		15	4		

TEMP1 Cr	CT1
3	1
10	1
8	2
5	0
15	3
12	0

/* Steps 1' - 3' */

TEMP3 Cr	CT2	TEMP4a Cr	Cm	TEMP4b Cr	CR2
3	0	3	78	3	1
10	0	10	78	10	1
8	0	5	74	5	1
5	0	12	74	12	1
15	0				
12	0				

TEMP3 Cr	CT2
3	1
10	1
8	0
5	1
15	0
12	1

The computed result in step 4 finally becomes: 10, 8, 5, and 15, which is the desired one.

Now we want to discuss what the mentioned BY-list optimization is about and why it fails for correlation queries with a correlation level greater than 1.

Let us reconsider the recent processing of Q3' and take a closer look at the modified query after phase B is finished. To repeat, this query is

```
RETRIEVE I.Ck
WHERE I.Ch = COUNT( J.Cm BY I.Cr
                    WHERE J.Cn = T1.CT AND J.Cm < 80
                    AND  I.Cr = T1.Cr )
```

At this point Ingres would attempt to eliminate some logically superfluous range variables from the subquery inside the COUNT according to the following reasoning: Knowing that T1.Cr is a complete projection of I.Cr (with duplicates removed), I.Cr is replaced by T1.Cr unless this would yield semantically incorrect transformations, with respect to the considered subquery inside the COUNT. In the given situation this procedure (called BY-list optimization) would produce:

```
RETRIEVE I.Ck
WHERE I.Ch = COUNT( J.Cm BY T1.Cr
                    WHERE J.Cn = T1.CT AND J.Cm < 80
                    AND T1.Cr = T1.Cr )
```

Thereafter this query is rescanned for trivial clauses like T1.Cr = T1.Cr which are dropped. Thus finally this procedure produces as input for phase C the following query:

```
RETRIEVE I.Ck
WHERE I.Ch = COUNT( J.Cm BY T1.Cr
                    WHERE J.Cn = T1.CT AND J.Cm < 80 )
```

The gained result for TEMP3 is of course the same as before. However the modified query for phase D now would become:

```
RETRIEVE I.Ck
WHERE I.Ch = T3.CT2 AND T1.Cr = T3.Cr
```

In this way one crucial link between Ri and TEMP3 would get lost with the undesired effect that Ingres actually delivers as result: 10, 8, 5, 15 and 12.
Consequently, the application of the Ingres BY-list optimization produces inconsistent results for correlation levels greater than one. Fixing this bug can be easily done, because for nested aggregates a range variable replacement in the BY-list optimization must only make sure that the variable in question does not appear in an outer BY-list.

4. CONCLUSION

In this note we showed that for SQL-like correlation queries involving the COUNT aggregate the transformation algorithms described in the literature fail to yield consistent results for some cases. For the remaining correlation queries (using MAX, MIN, SUM, AVG aggregates) those transformation algorithms are consistent, provided the proper use of NULL-values. On the other hand, the algorithms used to process analogous QUEL-like queries are reported to give the desired answers at any time, up to a minor repairable bug. Whether the SQL transformations can be adjusted in an elegant way relying on the current semantics of WHERE ... GROUP BY clauses is however questionable due to some inherent semantical deficiencies. To demonstrate this, a query given in [CHA76] for the well-known employee paradigm is reviewed:

```
SELECT DNO FROM EMP
WHERE JOB = 'CLERK'
GROUP BY DNO HAVING COUNT(*) > 10
```

The meaning of this query is supposed to be:
List the departments that employ more than ten clerks.
However, consider the slightly changed query where we ask for department that employ less than ten clerks. Now the current WHERE ... GROUP BY evaluation order fails to report departments that employ no clerks. In turn, QUEL allows a consistent formulation (OP stands for < or >):

```
RANGE OF E IS EMP
RETRIEVE ( E.DNO )
WHERE COUNT( E.NAME BY E.DNO WHERE E.JOB = 'CLERK' ) OP 10
```

In [KIM82] it is stated that the reason of the less-than-satisfactory performance of nested queries in existing relational database systems is that most types of nesting are not well understood. It remains to be added that these semantic transformations are indeed an important step towards efficiency for processing nested queries. However, the well-known fact that aggregates do not fit well into relational algebra has been emphasized by illustrative examples. Concerning the abstract Group-By operator an interpretation as generalized projection is given in [GRA81]. Doubtlessly further works needs to be done to integrate aggregates more smoothly into relational algebra. Then the design of query evaluation algorithms relying on query transformation will become a more reliable and powerful tool for efficiently processing very complex queries.

Literature:

[CHA76] D.D. Chamberlin, et al.:
SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control, IBM J. Res.&Dev., Vol.20, No.6, Nov. 1976, pp. 560-575.

[EPS79] R. Epstein:
Techniques for Processing Aggregates in Relational Database Systems, UC Berkeley 1979, Memo No. UCB/ERL M79/8.

[GRA81] P.M.D Gray:
The "Group by" Operation in Relational Algebra, Deen and Hammersley, Databases 1981.

[KIE83] W. Kiessling:
Database Systems for Computers with Inteliigent Subsystems, Ph.D. thesis, Techn. Univ. Muenchen, July 1983 (in German).

[KIE84] W. Kiessling:
Tuneable Dynamic Filter Algorithms for High Performance Database Systems, Proc. Intern. Workshop on High Level Computer Architecture, Los Angeles, May 21-25 ,1984, pp. 6.10 - 6.20.

[KIM82] W. Kim:
On Optimizing an SQL-like Nested Query, ACM TODS, Vol. 7, No. 3, Sept. 1982, pp. 443-469.

[MAK81]
A. Makinouchi, et al.:
The Optimization Strategy for Query Evaluation in RDB/V1, Proc. VLDB Cannes 1981, pp. 518-529.

[STO80] M. Stonebraker:
Retrospection on a Database System. ACM TODS, Vol. 5, No. 2, June 1980, pp. 225-240.