ON SEMANTIC REEFS AND EFFICIENT PROCESSING OF CORRELATION QUERIES WITH AGGREGATES

Werner Kiessling

Technische Universitaet Muenchen
Institut fuer Informatik
Arcisstr. 21
D-8000 Muenchen 2, West-Germany

Abstract

Recent transformation algorithms for speeding up processing of nested SQL-like queries with aggregates are reviewed with respect to the correctness of 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 uniform way to do these transformations efficiently and correctly under all circumstances. Also the algorithms for QUEL are reexamined regarding their correctness. It is shown that for a specific subset of QUEL-queries with aggregates a clearer semantics can be associated. Finally, benchmark results for Ingres show that considerable performance advantages may be gained for such query types by using dynamic filters. The consequence of all these observations is that more research is required to integrate correlation queries with aggregates into a unified operator tree model.

1. INTRODUCTION

This paper is concerned with a specific type of nested queries, involving aggregates and correlated predicates in a nested query block. A correlated predicate is a predicate in a nested block, which references a relation in an outer block. As an example consider the following sample relations and SQL-query ([CHA76]):

This report was prepared while the author was on leave at the University of California, Berkeley, CA 94720.
Die Arbeit wurde mit Unterstuetzung eines Stipendiums des Wissenschaftsausschuss der NATO ueber den DAAD ermoeglicht.

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 )
AND 3 ≤ PNUM AND PNUM ≤ 11
```

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 quantities on hand equal the highest quantities of those parts shipped before 1-1-80 and whose part number is between 3 and 11.

A similar example can be found in [KIM82]. In that paper a terminology for nested SQL-like queries is developed. We will 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 (SUPPLY.PNUM = PARTS.PNUM in our example) and some 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]. [LOH84] discusses the processing of nested queries in a distributed environment. In [KLU82] special access paths are described that may accelerate aggregate evaluation.

The standard method for evaluating correlation queries with aggregates is by *nested-iteration*, i.e. by evaluating the inner query block once for each substituted correlation value (PARTS.PNUM-value in our case). Furthermore, the semantics of a correlated SQL-query are defined in a convenient way by this nested-iteration procedure. Unfortunately, for a large set of such queries and database characteristics this method suffers from poor performance. Obviously, to design faster algorithms one has to look for ways where the inner query block, calculating the aggregates, may

be evaluated only once. This has been recognized at several sites independently and has led to the design of different algorithms to process correlation queries with aggregates, see [KIM82] and [KIE83] for SQL-like queries and [EPS79] for QUEL-like queries. In the first two works the nested-iteration semantics are the starting point from which equivalent transformation algorithms can be developed. The last work entirely defines the semantics of QUEL-queries using aggregates by the given evaluation algorithm. As an additional means to speed up the processing of correlation queries with aggregates, the utilization of dynamic filters, as described in [KIE84], is desirable.

However, these semantic transformations must be treated very carefully to give results consistent with the original semantics, defined by nested-iteration. In section 2 we will discuss the proper choice of defaults for aggregates over empty sets. As it will turn out, the solutions given for SQL fail for some queries using the COUNT aggregate. Unfortunately those algorithms cannot be easily adjusted to produce the desired answers. In section 3 the algorithms given in [EPS79] for processing analogous QUEL-queries are examined and shown to yield the desired results (modulo some repairable bugs, causing unexpected results in some cases). Since SQL and QUEL are widely spread, these results should be of immediate interest for a large user community. (In fact, the term *semantic reefs* was coined because a lot of "navigation" around all those bugs and inconsistencies was required, until matching results were established for untransformed queries, as being processed by the database system itself, and explicitly transformed queries, which were supposed to execute faster.) In section 4, QUEL-benchmarks for RTI-Ingres ([RTI83]) are reported. These performance measurements show the considerable gains to be expected when dynamic filters are applied. To produce verifiable and realistic results the synthetic database described in [BIT83] was used. Consequences asking for further research are outlined in section 5.

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
	2	3
	12	7

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
	2	3	6-2-79
	12	7	12-6-77

Evaluating Q1 by nested-iteration proceeds here as follows: Fetch each PARTS tuple, extract its PNUM value, test the restriction on PNUM and -if it passes the test- substitute it into the inner query block where it replaces PARTS.PNUM. Thereafter the inner block is evaluated and its result is compared to QOH of that PARTS tuple in question.
Doing so, the question of how to handle aggregates over empty sets arises immediately. Let us distinguish two cases:

(a) The default value for aggregates over empty sets is set to zero.[1]

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 an "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 not to satisfy the WHERE clause if the overall truth value is "false" or ?. [2]

Result: PARTS.PNUM
10

Now we transform query Q1 according to the algorithms described in [KIM82]. This transformation results in materializing a temporary relation TEMP and a subsequent join.[3]

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

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

[2] In [ZAN84] an illustrative treatment of the logical problems for dealing with NULL values can be found.

[3] Following [KIM82] the whole transformation requires two steps. First, the type-JA query is transformed into a so-called type-J query by the NEST-JA algorithm. Then an algorithm termed NEST-N-J transforms this type-J query into the join.

(b)
```
SELECT PNUM
FROM PARTS, TEMP
WHERE PARTS.QOH = TEMP.MAXQUAN
AND   PARTS.PNUM = TEMP.SUPPNUM
AND   3 ⩽ PARTS.PNUM
AND   PARTS.PNUM ⩽ 11
```

Informally, the processing of this algorithm can be summarized as follows: In the first step (a) the aggregate MAX(QUAN) is computed for each distinct SUPPLY.PNUM value. The second step (b) establishes the correspondence between the correlated PNUM values and evaluates the outer restriction on QOH.
For the materialization of the temporary relation TEMP recall the semantics of SQL-queries with a WHERE and a GROUP BY clause ([CHA76]): First the WHERE clause is applied to qualify tuples; then the respective groups are formed; then an aggregate function is applied to each group. According to these semantics step (a) gives:

TEMP	SUPPNUM	MAXQUAN
	3	4
	10	1
	2	3
	12	7

The final result of our query Q1, after step (b) is:

Result:	PARTS.PNUM
	10

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

The transformations described in [KIE83] are similar to the ones above, but the expected performance gains are even much more promising due to the use of dynamic filters. For the given example the final step (b) is identical, however step (a) is augmented as shown below:

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

In this modified form a *dynamic filter* $\Phi[PNUM]$ is conjunctively added, aiming the reduction of the size of temporary relations as much as desirable. Dynamic filters are predicates which are derived from already computed results during the evaluation of a query. The intention behind the determination of dynamic filters is to use them for a dynamic query modification, e.g. by a conjunctive addition to a restriction as above.
More formally, let X be a subset of the domain dom(R, r) of attribute r from R. A *filter* for X is a predicate $\Phi_X[r]$ in the variable r with the following property:

$$\forall\ x \in dom(R,r)\ :\ x \in X\ \rightarrow\ \Phi_X[r](x)$$

Now $\Phi_X[r]$ is called *dynamic filter*, if X is a subset of the projection $\pi_r(R)$.
For demonstration purposes we choose a so-called Min-Max filter for the set of relevant PNUM correlation values.
Let X be a subset of the domain of an attribute r. The *Min-Max filter* for X is the following predicate $\Phi_X^{mM}[r]$ in the variable r:

$$\Phi_X^{mM}[r] \equiv \text{`}\ \min\{\ x \mid x \text{ in } X\ \} \leqslant r \leqslant \max\{\ x \mid x \text{ in } X\ \}\ \text{'}$$

Then the Min-Max filter for the relevant PNUM correlation values is the following predicate:

$$\begin{aligned}\Phi_X^{mM}[PNUM] \equiv\ & \text{`}\ \min\{\text{PNUM} \mid 3 \leqslant \text{PARTS.PNUM} \leqslant 11\} \\ & \leqslant \text{PNUM} \leqslant \\ & \max\{\text{PNUM} \mid 3 \leqslant \text{PARTS.PNUM} \leqslant 11\}\ \text{'} \\ \equiv\ & \text{`}\ 3 \leqslant \text{PNUM} \leqslant 10\ \text{'}\end{aligned}$$

The impact of attaching this dynamic filter to the query is the reduction of the cardinality of TEMP as shown below:

TEMP	SUPPNUM	MAXQUAN
	3	4
	10	1

We will return to the performance gains for correlation queries with aggregates when we apply dynamic filters in section 4, where RTI-Ingres is benchmarked. For the subsequent discussion however, the use of dynamic filters makes no difference, therefore we will refer to Kim's basic transformation algorithm.

2.2. Troubles with the COUNT-aggregate

Until now everything seems to work nicely, assuming the proper use of NULL-values that assures the equivalence of the transformed query to the nested-iteration semantics. There seems to be some trouble with this semantic transformation only if aggregates over empty sets are treated inadequately. But there are examples where there is no straightforward way out of this dilemma of aggregates over empty sets and therefore this type of semantic transformation is incorrect for some type-JA SQL-queries. To shown this, we will evaluate a query Q2 that 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 zero. (This also implies 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 )
AND 3 ⩽ PNUM AND PNUM ⩽ 11
```

The meaning of query Q2 is supposed to be.[4]
Find the part numbers of those parts, whose quantities on hand equal the number of shipments of those parts before 1-1-80 and whose part number is between 3 and 11.

Evaluating Q2 according to the nested-iteration semantics yields:

Result: PARTS.PNUM
10
8

The transformation of Q2 using Kim's algorithms gives:

```
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
AND   3 ≤ PARTS.PNUM
AND   PARTS.PNUM ≤ 11
```

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

TEMP'	SUPPNUM	CT
	3	2
	10	1
	2	1
	12	1

Result: PARTS.PNUM
10

Again the results differ. But now we cannot establish a match, because COUNT is a totally defined function. The reason, why this transformation fails in this particular case is the following: because of the WHERE...GROUP BY semantics, materializing TEMP' eliminates absent SUPPLY.PNUM values; thus these empty sets are not counted and not evaluated to CT = 0. On the other hand, the nested-iteration method does count empty sets. 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.[5]
For completeness, it also should be mentioned that [JAR82] addresses a related problem with empty relations, which arises if transformations involving queries with existential quantifiers are attempted.

How to fix these bugs?
If one does not want to resort to a different algorithm then the following modification comes into mind:
Trial correction:
Adjust the transformation algorithm for correlation queries with COUNTs by a posteriori recovering lost aggregates over empty sets in the following way:
TEMP' is defined as before, however the join 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' ) )
AND  3 ≤ PARTS.PNUM
AND  PARTS.PNUM ≤ 11
```

Adding the OR clause preserves those tuples of PARTS, that have no matching PNUM value in SUPPLY. This construct is a one-sided outerjoin (see, e.g., [ROS84]), with an additional restriction on QOH testing for 0 (= COUNT(empty set)). Unfortunately this solution only works for 1-level deep correlated type-JA queries like our Q2. The following example with a correlation depth of 2 shows that in general the transformation under consideration cannot be fixed for arbitrarily correlated SQL-queries with COUNTs. (Lower case letters denote attributes belonging to a relation denoted by a respective upper case letter.)

```
SELECT r1
FROM  R
WHERE r2 = ( SELECT aggr(s1)
             FROM  S
             WHERE s2 =
               ( SELECT COUNT(t1)
                 FROM T
                 WHERE t2 = r3
                 AND restriction(t3) ) )
```

Here, r3 in the innermost query block correlates to relation R in the outermost block. Applying the considered transformations would produce two temporaries. The sketched method of information preservation by using outerjoins can retain r3-values not matched by any t2-value. But we are unable to deal with the following case: If for some non-matched r3-value s2 happens to be 0, then aggr(s1) in the middle query block yields some arbitrary value, depending on the current database contents. That's why the required additional restriction on r2 for the outerjoin cannot be attached automatically. A detailed example for this type of inconsistency can be found in [KIE84b].
Consequently, to devise a transformation that will produce

[4] Whether this is a question of practical interest is irrelevant for our discussion.

[5] The loophole in [KIM82] lies in the proof of lemma2 upon which the 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 C1 column has the same value as the Cp column of the Ri tuple,...*"
(In our example the roles of Ri, Rt, C1 and Cp are occupied by PARTS, TEMP', PARTS.PNUM and TEMP'.SUPPNUM.)

consistent results for any type-JA query, different algorithms must be considered.

3. REVISITING QUEL SEMANTICS FOR AGGREGATES

As mentioned in the introduction, the semantics for QUEL ([STO76]) queries with aggregates are not conveniently defined by nested-iteration. The reason is probably founded in the following differences of QUEL, as compared to SQL. Unlike SQL, QUEL distinguishes between scalar 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 )
AND  3 ≤ P.PNUM AND P.PNUM ≤ 11
```

However, in this query the nested 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 is termed a *scalar aggregate* , and it evaluates to a single value that 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 so-called *BY-list*. Aggregates with a BY-list are termed *aggregate functions*. In fact, the correct choice can be made only if one is aware of all the implementation details, which of course cannot and should not be expected from a QUEL user. The correct way is to put the correlation attribute of the outer relation in the BY-list, as it is depicted 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 )
AND  3 ≤ P.PNUM AND P.PNUM ≤ 11
```

Because of this freedom of explicit bindings through BY-lists a much broader class of aggregate queries than in SQL can be defined. Therefore, a comprehensible nested-iteration semantics cannot be assigned to all nested queries with aggregates in QUEL. The semantics are only defined by the evaluation procedure described informally in [EPS79]. If applied to our query Q2', 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 inner query with aggregate function locally,
maintaining the connection between the inner correlation
column values and their respective aggregate value. */
/* Be careful 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
AND  3 ≤ P.PNUM AND P.PNUM ≤ 11
```

According to this algorithm Q2' gets processed as follows:

/* Steps 1 - 3 */

TEMP1	PNUM	CT
	3	0
	10	0
	8	0
	2	0
	12	0

TEMP2a	PNUM	SHIPDATE
	3	7-3-79
	3	10-1-78
	10	6-8-78
	2	6-2-79
	12	12-6-77

TEMP2b	PNUM	CT
	3	2
	10	1
	2	1
	12	1

TEMP1	PNUM	CT
	3	2
	10	1
	8	0
	2	1
	12	1

Result: 10, 8.

Remarks:
(1) Note that projecting initially all outer correlation values into a temporary guarantees to lose no non-matched correlation value, as opposed to the SQL transformation. 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.[6] If however, for whatever reasons, there exist duplicate values in the outer correlation column then this algorithm fails to be equivalent to nested-iteration. (On the contrary, this issue does not arise for the SQL transformations.) 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 queries involving aggregates other than COUNT this algorithm is slower compared to the SQL transformations, because it requires two joins (step 2 and 4). However, it is capable of processing a larger class of correlation queries that 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, while it is not directly applicable to those in [KIM82].
Besides revealing several bugs for QUEL, which went undetected for a long period, an important result of these observations is that we are now able to assign some clearer semantics to a certain subclass of QUEL-queries with aggregates. Namely, if we consider the class of all QUEL-queries that we get by translating type-JA SQL-queries into their QUEL-counterparts with the proper BY-list choice, then the algorithm of [EPS79] implements the nested-iteration semantics (up to the mentioned exceptions).

[6] For Ingres this proposition actually holds for 1-level nested queries. For correlation depths greater than 1 a procedure called "BY-list optimization" produces erronous results in some situations (see [KIE84b] for an illustrative example).

4. INGRES BENCHMARKS UTILIZING DYNAMIC FILTERS

In this section we will provide benchmark results for some QUEL correlation queries with aggregates for RTI-Ingres ([RTI83]). Hereby, the performance of processing the original query in RTI-Ingres is compared to that of the explicitly transformed version according to [EPS79], augmented by dynamic filters. The first query schema to be tested is stated below.

Queryschema 1:
```
RETRIEVE ( R.r1, R.r2 )
WHERE R.r5 ≤ const1
AND   R.r4 < avg( S.s1 BY R.r5
                  WHERE S.s3 ≤ const2
                  AND R.r5 = S.s2 )
```

The equivalent transformed schema is the following:

```
(a)
RETRIEVE INTO TEMP1 ( R.r5, aggval = 0 )
WHERE R.r5 ≤ const1                          (*)
```

Note: This step must remove duplicates. This can be achieved by choosing a proper storage structure for TEMP1, which eliminates duplicates.

```
(b)
RETRIEVE INTO TEMP2a ( R.r5, S.s1)
WHERE S.s3 ≤ const2
AND   R.r5 = S.s2
AND   Φ[r5]                                  (**)
```

Note: This step requires duplicates to be preserved. Again, this must be communicated to Ingres by selecting a proper storage structure.

```
(c)
RANGE OF Tp1 IS TEMP1
RANGE OF T2a IS TEMP2a
RETRIEVE
INTO TEMP2b ( T2a.r5,
              aggval = avg( T2a.s1 BY T2a.r5 ) )

(d)
RANGE OF T2b IS TEMP2b
REPLACE Tp1 ( aggval = T2b.aggval )
WHERE Tp1.r5 = T2b.r5

(e)
RETRIEVE ( R.r1, R.r2 )
WHERE R.r4 < Tp1.aggval
AND   R.r5 = Tp1.r5
```

Comments to the transformed schema:
(*): This is simply an application of the well-known strategy of shifting operators down an operator tree. Ingres (at

least the University version) is not capable of doing this, because the entire query evaluation process is separated into an initial phase, where all aggregates are processed and replaced by their results, and a subsequent aggregate-free processing phase. Indeed, this separation reflects the fact that the query evaluation model is not unified. We will come back to this observation in the concluding section.
(**): Here the idea of utilizing dynamic filters is applied. A dynamic filter $\Phi[r\,5]$, characterizing the relevant correlation values for r5, is conjunctively added. The goal is of course to reduce the cardinality of TEMP2a. We will explain our choice of a concrete filter type after having presented the second benchmarked query schema.

Queryschema 2:
RETRIEVE (R.r1, R.r2)
WHERE R.r5 $\leqslant$ const1
AND R.r4 < avg(S.s1 BY R.r5
WHERE S.s3 $\leqslant$ const2
AND S.s2 = T.t1
AND R.r5 = T.t2)

Note that schema 2 is similar to schema 1, except that the correlation term R.r5 = S.s2 is replaced by the more complicated expression S.s2 = T.t1 AND R.r5 = T.t2.
The transformed version, augmented again by a dynamic filter, is as follows:

(a) As in query 1 transformation.
(b) RETRIEVE INTO TEMP2a (R.r5, S.s1)
WHERE S.s3 $\leqslant$ const2
AND S.s2 = T.t1
AND R.r5 = T.t2
AND $\Phi[r\,5]$

(c), (d), (e) As in query1 transformation.

Now let us turn to describe the choice of a specific filter type for $\Phi[r\,5]$. The idea behind the flexibility offered by dynamic filters is that the selection of a proper filter type is tuneable to the performance characteristics of the database architecture in question. For a conventional database system like Ingres the application of a total filter ([KIE84]) looks most promising. The *total filter* for a subset X of the domain of an attribute r is the following predicate in r:

$$\Phi_X^{tot}[r] \equiv \text{`} \bigvee_{x \in X} (r = x) \text{'}$$

So the total filter for the relevant r5-correlation values is defined as:

$$\Phi_X^{tot}[r\,5] \equiv \text{`} \bigvee_{x \in X} (R.r5 = x) \text{'},$$

where X = { R.r5 | R.r5 $\leqslant$ const1 }

As a lucky chance, the set X is our temporary TEMP1, which is computed anyway. Thus this total filter can efficiently be simulated in QUEL by making the following choice:

$$\Phi_X^{tot}[r\,5] \equiv \text{`} R.r5 = Tp1.r5 \text{'}$$

Benchmark description:
The benchmarked database was the synthetic database of [BIT83], which we found to be a very helpful test tool. The names of relations and attributes are self-explanatory, only the attributes needed are listed below. (E.g. onektup is a relation of cardinality 1000, hundreda is an attribute whose domain ranges from 1 to 100)

Relations:
onektup (hundreda, thousanda, unique1a,
unique2a, stringu1a, . . .)
twoktup (hundredb, thousandb, unique1b, . . .)
fivektup (hundredc, thousandc, . . .)

We made the following assignments for the relations and attributes appearing in query schema 1 and 2:

R = onektup, S = twoktup, T = fivektup
r1 = . . . , r2 = stringu1a, r3 = unique2a,
r4 = thousanda, r5 = . . .
s1 = thousandb, s2 = hundredb, s3 = unique1b
t1 = hundredc, t2 = thousandc

The following benchmark series were performed on the database with no useful storage structures and secondary indexes available:

For queryschema 1:
Series 1.1: r1 = unique1a, r5 = hundreda
Series 1.2: r1 = hundreda, r5 = unique1a
For queryschema 2:
Series 2.1, series2.2 with similar correspondences for r1 and r5.

The benchmarks were run on a VAX/780 in multiuser mode. The performance results are depicted in the appendix, each series run for several const1, const2 values. The measurements were obtained by using RTI-Ingres' performance monitoring facilities. Besides the consumed CPU time and totally elapsed time, the cardinality of the final result is listed as well. As can be observed from these tables, there are huge gains for the transformation algorithms with dynamic filters. They outperform Ingres' processing algorithms up to a factor of 5 for schema 1 and up to a factor of 10 for the complex schema 2. Clearly dynamic filtering achieves a higher speed-up for more complex queries. Note also that the transformations were coded in EQUEL ([STO76]), which introduces a considerable performance penalty. Thus dynamic filters are shown not only to be a useful tool for join processing ([KIE84]), but also for correlation queries. Moreover, the set-oriented transformation approach, which materializes several temporary relations, proves to be also viable in the context of a conventional, non-distributed database system. Also, it should be recalled that Kim's transformations can likewise be ameliorated by dynamic filters, which will give another performance gain in addition to what is indicated in [KIM82] compared to nested-iteration processing.

In conclusion, let us state one more observation. For some variations of the complex schema 2 we encountered situations where the results of the supposedly equivalent query schema and those of the transformed version differed in so far as the original schema delivered one tuple less in the result. These mismatches must be contributed to the duplicate problem. The preservation of duplicates for a subsequent aggregate computation cannot be guaranteed by the query processor for complex queries, and schema 2 is an

example for that. So again a striking example showed up that aggregates requiring duplicate preservation do not fit well into relational algebra, at the time being. It is suspected that similar problems exist not only for this benchmarked database system, but also for other products.

5. CONCLUSION

In this paper we showed that for SQL-like correlation queries involving the COUNT aggregate the transformation algorithms described in the literature fail to yield correct results for some cases. For the remaining correlation queries (using MAX, MIN, SUM, AVG aggregates) those transformation algorithms are consistent with the desired semantics, if NULL-values are used. On the other hand, the algorithms used to process analogous QUEL-like queries are reported to give the desired answers (up to some repairable bugs), at the expense of less efficiency for non-COUNT aggregates. In summary, the question, whether the discussed transformations of type-JA queries are equivalent to nested-iteration, is answered in the next table.

	aggr ≠ COUNT	COUNT
SQL on SystemR	yes	no
QUEL on Ingres	currently no, yes, if NULLs are supported	yes

Whether the SQL transformations can be adjusted in an elegant way relying on the current semantics of WHERE ... GROUP BY clauses is questionable because of some inherent semantical problems. 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.

Now consider the slightly changed query where we ask for departments that employ less than ten clerks. This time the current WHERE ... GROUP BY evaluation order fails to report departments that employ no clerks. In turn, QUEL allows a consistent formulation, reporting also departments without clerks.

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

In [KIM82] it is stated that the reason for the less-than-satisfactory performance of nested queries in existing relational database systems is that most types of nesting are not well understood. It should be added that these semantic transformations are indeed an important step towards efficiently processing nested queries, especially if dynamic filters are applied in addition. The reported benchmarks demonstrate that impressive performance improvements are achievable. However, the well-known fact that aggregates do not fit well into relational algebra has been pointed out by illustrative examples. Doubtless, further work needs to be done to integrate aggregates more smoothly into relational algebra; e.g. in [ROS84] it is stated that a compact operator tree model covering the query class considered is still lacking. Then the design of query evaluation algorithms relying on query transformation will become a more reliable and powerful tool for efficiently processing complex queries. This is mandatory if complex applications such as expert systems, built on top of a database system (see e.g. [STO84]), are to be implemented sufficiently fast and trustworthy.

Acknowledgment: I wish to thank Yannis Ioannidis for carefully reading a draft of this paper. Likewise I am grateful to Michael Stonebraker for providing me the opportunity to spend my scholarship at UC Berkeley.

Literature:

[BIT83] D. Bitton, D.J. DeWitt, C. Turbyfill:
Benchmarking Database Systems: A Systematic Approach, Proc. VLDB Florence, 1983, pp. 8-19.

[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.

[JAR82] M. Jarke, J.W. Schmidt:
Query Processing Strategies in the Pascal/R Relational Database Management System, Proc. SIGMOD 1982, pp. 256 - 264.

[KIE83] W. Kiessling:
Database Systems for Computers with Intelligent Subsystems: Architecture, Algorithms, Optimization, techn. report TUM-I8307, Aug. 1983, Techn. Univ. Munich (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.

[KIE84b] W. Kiessling:
SQL-like and QUEL-like Correlation Queries with Aggregates Revisited, UC Berkeley 1984, Memo No. UCB/ERL/84/75.

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

[KLU82] T. Klug:
Access Paths in the 'Abe' Statistical Query

Facility, Proc. SIGMOD 1982, pp. 161 - 173.

[LOH84] G.M. Lohman, et al.:
Optimization of Nested Queries in a Distributed Relational Database, Proc. VLDB Singapore 1984, pp. 403-415.

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

[ROS84] A. Rosenthal, D. Reiner:
Extending the Algebraic Framework of Query Processing to Handle Outerjoins, Proc. VLDB Singapore 1984, pp. 334-343.

[RTI83] Relational Technology Ingres
EQUEL/C User's Guide Version 2.0 VAX/UNIX, Jun. 1983.

[STO76] M. Stonebraker, E. Wong, P. Kreps:
The Design and Implementation of INGRES, ACM TODS, Vol. 1, No. 3, Sept. 1976, pp. 189-222.

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

[STO84] M. Stonebraker, et al.:
QUEL as a Data Type, Proc. SIGMOD 1984, pp. 208 - 214.

[ZAN84] C. Zaniolo:
Database Relations with Null Values, Journ. of Comp. and Sys. Sc., Vol. 28, No. 1, Febr. 1984, pp. 142-166.

Table A.1: Benchmark series 1.1.

const1	const2	card result	cpusec RTI	cpusec transfo	elapsedsec RTI	elapsedsec transfo
100	1200	507	184	45	375	111
60	1200	302	177	31	384	79
30	600	147	98	20	256	54
15	300	64	58	17	165	63
5	100	9	32	15	82	33

Table A.2: Benchmark series 1.2.

const1	const2	card result	cpusec RTI	cpusec transfo	elapsedsec RTI	elapsedsec transfo
600	1200	49	66	46	180	124
300	600	52	43	30	116	86
150	300	47	31	23	54	66
50	100	12	24	16	66	40

Table A.3: Benchmark series 2.1.

const1	const2	card result	cpusec RTI	cpusec transfo	elapsedsec RTI	elapsedsec transfo
100	1200	496	790	109	1418	245
60	1200	304	784	75	1650	199
30	600	148	432	46	1459	197
15	300	71	251	36	1047	132
5	100	21	118	31	363	71

Table A.4: Benchmark series 2.2.

const1	const2	card result	cpusec RTI	cpusec transfo	elapsedsec RTI	elapsedsec transfo
600	1200	301	2916	920	10703	2514
300	600	149	1429	132	2388	340
150	300	78	745	68	1392	158
50	100	21	302	39	1083	101