Duplicates and Translation of Nested SQL Queries into XRA

N.Th. Verbrugge
CWI
Kruislaan 413, 1098 SJ Amsterdam

Abstract

The PRISMA/DB system[1] contains a parser to translate the database language SQL into eXtended Relational Algebra (XRA). The early definition of XRA, which has a multi-set semantics, proves inadequate for translating SQL according to its nested-iteration semantics. The prime cause is that no distinction is made between original and generated duplicate tuples during nested query handling. To achieve a correct translation, tuple identifiers were introduced into XRA and the system.

Keywords: Duplicates, SQL, XRA, nested query evaluation, nested-iteration method, tuple identifiers, syntax-directed SQL translation.

1 Introduction

The PRISMA/DB system [Kersten] supports two query languages, SQL and PRISMALOG (a logical language). SQL [SQL] was chosen since it is the standard database query language, and PRISMALOG adds knowledge base characteristics through its inference techniques. Both languages are translated into the internal language of the PRISMA/DB system, an extended relational algebra called XRA [Wilschut]. The XRA structure is then passed to the Query Optimizer, which is responsible for deriving an optimal query execution schedule.

Our primary goal is to develop a general translation heuristic, or even better, a syntax-directed translation from SQL to XRA.

It is striking that many systems do not support the full power of SQL, especially its nesting capabilities. These systems avoid nested queries for several reasons: (1) the asymmetry of the query biases the query optimizer to a nested query handling strategy, using for-loops which may not be optimal, (2) translation problems concerning duplicates (tuples of a table with equal values for all their attributes) in relational algebra translations of SQL, and (3) block notation is hard to learn for non-programmers ([Astrahan], referring to psychological studies).

In this paper, we present a solution to the first two problems, i.e. we present a correct translation of a nested SQL query into XRA. This translation distinguishes duplicate tuples by using artificial primary keys for relations, called tuple identifiers. Except maybe for handling "unknowns", which is *the* other SQL problem [Date2], nested queries can always be handled correctly now.

The asymmetry is removed by translating the nested predicate to a (symmetric) join. The join can be optimized taking the relative sizes of the joined relations into account. Other opportunities for query optimization in relational algebra are described by [Ullman] and [Hall].

Concerning the third reason to avoid nesting, in our opinion, nested queries can be quite useful for programming SQL, as e.g. [Date] shows. While we do not fully deny block notation complexity,

[1]The research for this article is conducted in the context of the PRISMA project, which is supported by the Dutch "Stimuleringsprojectteam Informaticaonderzoek" (SPIN).

nesting may ease query formulation by users considerably. A system should at least provide for the option, letting the user decide whether to nest the query or not.

The next section explains the multi-set semantics of SQL and XRA. Before we present our solution, we first show that XRA cannot be used to translate SQL. This is primarily caused by the semantics of nested queries, which require a distinction to be made between generated and original duplicates in relations. In section 4, we give the solution to the translation problem: tuple identifiers. Tuple identifiers are not only implementation handles to ease database system construction, they also prove to be essential to support multi-set oriented processing of nested queries. The general characteristics of the correct nested predicate translation which uses tuple identifiers in XRA are discussed, and an example translation is given. (The results propagate to SQL2 and SQL3 as well [SQL2].) A syntax-directed translation of the nested predicates into XRA is provided after that. It is the key to the complete syntax-directed translation of SQL into XRA with tuple identifiers. Section 5 presents our conclusions.

2 Nested query semantics

To understand the translation problems caused by nested queries, the main characteristics of SQL and XRA concerning the translation process and the key role of duplicates must be understood. Therefore, the semantics of SQL and XRA are explained, and a sample nested query translation is given.

2.1 SQL: the nested-iteration semantics

The database relations on which SQL operates are multi-sets, i.e. the set may contain duplicates. SQL queries typically contain *(sub)query blocks* of the well-known form:
SELECT-FROM-WHERE. A *nested (sub)query* in SQL is a (sub)query block containing one or more subquery blocks. The semantics of query and subquery blocks are equal.

We first give a simple example of an SQL nested query. It shows what the inner subquery and outer query blocks are. The database schema is the same as the schema of the suppliers-and-parts database by [Date]. It contains the relations S(snr, sname, status, city), P(pnr, pname, color, weight, city), and SP(snr, pnr, qty).

SQL query nr. 1:

```
SELECT sname
FROM S                                  outer block
WHERE S.snr IN      (SELECT snr
                     FROM SP            inner block
                     WHERE pnr = 1);
```

The query is handled by *conceptually* doing the following: for each tuple in the outer block, see whether it satisfies the search condition(s) given. We have to keep in mind that for every tuple, nested predicates (boolean expressions containing a subquery) may come out differently, depending on the tuple's own values. Therefore, the inner block may have to be evaluated again for every outer block tuple.

This way of (sub)query handling is called a *nested-iteration method.* It is applied recursively, and another formulation is the following:

```
FOR every outer block tuple (FROM S)
DO
        recursively evaluate the inner block(s);
        IF WHERE-condition
        AND HAVING-condition hold THEN
                add the outer block tuple to the result
        FI
OD;
apply aggregate functions and project the SELECT columns.
```

The outer block tuples are formed by taking the Cartesian product of the relations mentioned in the FROM-part. (Grouping can also be incorporated; this is not done here for simplicity.)

The result of the example query contains those S-tuples for which there is an SP-tuple with both "S.snr = SP.snr", and the "SP.pnr" value equals 1.

Most systems which allow nested SQL use a nested-iteration-like strategy for nested query handling. Query execution time can be long in these systems because of the asymmetry of the query. The SQL standard [SQL] leaves the evaluation strategy to the system implementor, as long as a system obeys the same semantics. This is what we do in PRISMA/DB: we translate SQL into XRA, which includes optimization techniques to improve system response time.

2.2 XRA: algebraic multi-set semantics

The database relations on which XRA operates are multi-sets as well. The basic operations in XRA are relational expressions, consisting of a relational operator and its arguments. The result of any relational expression has the form of a relation again. This relation may contain duplicate tuples (also called duplicates for short), and therefore is a multi-set. This shows that XRA has a multi-set semantics.

In XRA, a database relation is an elementary relational expression (*rexpr*), e.g. "SP". An example of a relational operation is "uniq(*rexpr*)", which removes the duplicate tuples from *rexpr*. Another operation, "select(%1 > 10, *rexpr*)", selects only those tuples from *rexpr* of which the first attribute's value[2] is greater than 10. Table 1 shows some of the XRA operations used in query translation, along with roughly corresponding SQL constructs.

Function	*SQL*	*XRA*
project columns from *rexpr*	SELECT	project(*plist, rexpr*)
condition must hold for tuple	WHERE/HAVING	select(*cond, rexpr*)
Cartesian product	FROM	cp(*rexprlist*)
relational join	FROM/nesting	join(*rexpr1, cond, rexpr2*)
eliminate duplicate tuples	DISTINCT	uniq(*rexpr*)
grouping and aggregate funcs	GROUP BY, MAX, etc.	gb(*grlist, funcs, rexpr*)
sort tuples	ORDER BY	sort(*sortlist, rexpr*)

Table 1. The function of some operations in SQL and XRA.

2.3 Simple nested query translation

In this section, we give a correct XRA translation of the aforementioned nested SQL query, *SQL query nr. 1*, provided that S does not contain duplicates. In this case, S is a set rather than a multi-set of tuples. Moreover, to give a simple translation, we also assume that S has the primary key snr.

[2]In the PRISMA/DB system, columns are referred to by their relative position number in the table or subexpression(s). Column number calculation is not essential to this article. Therefore, from now on the names are used instead of numbers.

Under these two assumptions, an XRA translation of SQL query nr. 1 using column names becomes:

Dupl. translation nr. 1key:

```
?project(S.sname,
        uniq(project((S.sname, S.snr),
              join(  S,
                     S.snr = SP.snr,
                     select(SP.pnr = 1, SP)))))
```

The *join* operation joins table S and SP. The combination *uniq-project* projects over the columns **sname** (the supplier's name) and snr (the supplier's number), and discards the duplicates. The result of the outermost *project* produces the names of the suppliers of parts with number 1, without discarding duplicates. The question mark denotes a query.

3 Duplicate problem analysis

As we will inductively show in this section, the current definition of XRA [Wilschut], is not expressive enough to translate SQL nested queries correctly. The problem concerns tuple duplicates and translation of nested predicates. The analysis naturally leads to an extra construct in XRA to control removal and retention of duplicates, i.e. the tuple identifier. The role of the tuple identifier in the XRA translation will be deferred to section 4.

Other authors have recognized this SQL processing problem as well. [Kim] and [Kiessling] worked on SQL unnesting, to ease query optimization. They did not develop a complete solution; otherwise, we could have used it. [Murali] also uses primary key knowledge, assuming its presence in each relation, to obtain a dataflow algorithm for nested query execution. [Ceri] made a relational algebra translation which is limited to tuple sets. Therefore, concerning duplicates, our XRA and their Relational Algebra differ as follows:

- Their Relational Algebra does not deal with duplicates, while our XRA does. Therefore, in their approach duplicates are always eliminated. Queries and subqueries are viewed as if they were always written "SELECT DISTINCT...". They pointed out that extensions to their Relational Algebra with control over duplicates are feasible.

- They only allow GROUP BY in a query if it applies to the outermost query block, whereas we allow grouping in all (sub)query blocks. This is a good decision in their case. Namely, control over duplicates in subqueries (and in relational algebra joins) is essential for correctly translating grouping and aggregate functions.

3.1 Duplicate elimination and preservation

Why is it so important that the result of a nested predicate evaluation (and therefore a nested query) in XRA complies with the semantics of SQL? It is needed for aggregate evaluation.

For example, consider a query in which the original relation was O = {a, a, b}, where a and b are tuples, and a has a duplicate since it occurs twice. Suppose also that a nested WHERE search condition and a GROUP BY operate on O, and the result of the former is incorrect, e.g. {a, a, a, a}. Then the XRA gb-operation in combination with e.g. a SUM aggregate function will have a result which is two times the SQL result if the SUM is calculated over a non-zero column of the a-tuples. Having duplicates in an answer can be overcome easily by users, but faulty calculations of functions will generally not be detected by them.

The semantics of XRA expressions is defined such that the result of the (sub)expression is a multi-set of tuples. Duplicates can be eliminated from the (intermediate) result explicitly by applying the "uniq" operator. In SQL, duplicate tuples can be removed with the "DISTINCT" option in the

SELECT-list of a query block. Similarly, "UNION" removes duplicates, in contrast to "UNION ALL".

Less obvious is the fact that some kinds of nested SQL queries also require duplicate removal, while others require duplicate preservation. Notable is the use of aggregate functions in combination with grouping, of which we gave an example. Always removing duplicates would cause aggregates requiring duplicate preservation to behave wrongly, as Kiessling noted [Kiessling]. Always retaining duplicates would be a bad strategy for aggregates requiring duplicate elimination. However, even without aggregate functions and grouping, wrong results are easily produced. Actually, for correct duplicate handling, we must be able to distinguish between:

- "original" duplicates, which were present in the original relation(s) used. They must be preserved in a result. And
- "generated" duplicates, which are formed by projections, or projections on joins (or Cartesian products). They must be removed from a result.

This problem concerning correct duplicate handling will be called the *duplicate problem.*

It may look as though *primary keys* can always serve to solve the duplicate problem, like in the translation in the previous section. This is not true, since some tables, such as intermediate tables and tables with duplicates, do not have primary keys. Next to that, especially if the primary key is a multi-valued key (i.e. contains multiple attributes [SQL]), we should answer the following questions: Do any primary key attributes participate in the intermediate result (=XRA subexpression)? Are they used in arithmetic or aggregate functions? Which of these attributes should then be eliminated from the result?

3.2 XRA cannot solve duplicate problem

We now want to show that XRA cannot translate SQL correctly if it has no tuple identifiers at its disposal.

For nested query translation, all XRA operators which have some effect on duplicates must be considered. They are: join, cp, project, uniq, gb, as well as (multi-set) union, intersection, and difference. The first five can be used for nested predicate translation, the last three for AND, OR, and NOT in search conditions containing one or more nested predicates. Nested predicate translation will be covered first. If nested predicates are not correctly translated, and thus the duplicate problem is not solved, then union, intersection, and difference for search conditions also yield incorrect results. Furthermore, we show why correct nested predicate translation is crucial to aggregate evaluation in nested queries.

A nested predicate in SQL has the form:

<nested predicate> ::=
<lvalexpr> <nesting compop> <subquery>

<subquery> ::=
(SELECT [DISTINCT] <valexpr>
FROM <table list>
[WHERE <where sc>]
[GROUP BY <grouping attrs>]
[HAVING <having sc>])

Nested predicate translation cannot be done correctly in XRA. Namely, in terms of XRA, a nested predicate in SQL requires:

1. a cp or join over the outer(O) and the inner(I) block tuples, including a condition which must hold. The O tuples come from the SQL query containing the <nested predicate>. The I tuples are the Cartesian product of the subquery's <table list> tables. The condition compares <lvalexpr> and <valexpr>.

2. sometimes: grouping and aggregate function calculation, and selections for WHERE and HAVING search conditions on the group or the aggregates.

3. extraction of the outer(O) tuples from the (possibly grouped) product of O and I.

4. that the duplicate problem must be solved. That is, the result should contain just as many duplicates for a tuple as O had, or none at all, because any search condition either holds for all tuple duplicates or for none.

Especially the last point cannot be accomplished. Namely, we can combine the XRA operators cp/join, project, uniq, or gb to form the result. The effects of cp and join are the same, except that a join also incorporates a search condition for the resulting tuples, and thus a tuple and its duplicates are either all kept or all removed. In general, cp/join and project generate duplicates and preserve generated duplicates. On the other hand, gb and uniq eliminate all duplicates of a tuple; one tuple is saved. To translate a nested predicate, we have to start out with a **join** of two or more multi-sets. (Join stands for cp as well here.) The join is needed because the predicate itself, and possibly a WHERE condition in the subquery as well, compare O-attributes with I-attributes. XRA can only compare attributes from O and I by means of a join, or by a combination of a Cartesian product cp and a selection.

For the second and further operations, we have more choice. Selections and other XRA operations which do not structurally change the number of duplicates for a tuple are not shown.

An important thing to keep in mind is that, in general, O and I both contain duplicates. Such duplicates cannot be removed from O or I by XRA in all cases. Namely, suppose we have an aggregate or GROUP BY over O or I *in the subquery*. Without the duplicates of O and I, a gb operator could return an incorrect result compared to SQL nested query semantics. Even if duplicates were first removed from O and/or I, by doing uniq(O) and/or uniq(I), duplicates of the O-tuples (and I-tuples) can be introduced again, that is generated, by the join. We are looking for a solution which works for all cases, since during query translation we cannot determine whether an intermediate result will contain duplicates relative to a subset of its columns (e.g. its O-tuple part). After the join, what can follow is:

- **project**: project(O-attributes, join(O, I)).
  Suppose for one O-tuple there were more than one joinable I-tuples, that is tuples which satisfied the join condition. Then the result is too large: it contains (extra) duplicates for the joinable O-tuples. A third operator could work correctly, but not in all cases:
  - **uniq**: uniq(project(O-attributes, join(O, I))).
    This works correctly only if O contains no joinable tuples with duplicates; this cannot be guaranteed. Otherwise, the result contains only one tuple out of a group of duplicate tuples from O, which is incorrect. [Ceri] also observed this, since set-semantics were used in their Relational Algebra. This is the same as always applying a uniq operator to every intermediate result in our XRA.
  - **join**: join(O, project(O-attributes, join(O, I))).
    This is a join again, which has the same structure as the join we started off with. Analysis of the cases therefore is done already.
  - **gb**: gb(O-attributes, O-aggregates, project(O-attributes, join(O, I))).
    This has the same shortcomings as uniq(project(O-attributes, join(O, I))).
- **gb**: gb(OI-attributes, OI-aggregates, join(O, I)).
  To extract the outer(O) tuples, a **project** should follow (after a select for some search condition was performed):
  project(O-attributes, gb(OI-attributes, OI-aggregates, join(O, I))).
  This has the same characteristics as project(O-attributes, join(O, I)).

- **join**: could have been incorporated in the first **join**.
- **uniq**: uniq(union(O, I)) serves no purpose for nested predicate calculation.

This shows that such a nested predicate translation cannot be duplicate correct. Note that an extra join and project with the original O (or I) will not help, again because the join generates duplicates. Also, deeper nesting through a subquery inside the inner block sometimes prohibits this, especially if it contains aggregate calculations. Moreover, multi-set union, intersection, and difference cannot be of help either, since the operands should be duplicate correct first. This also implies that AND, OR, and NOT in search conditions containing nested predicates cannot be translated according to SQL semantics.

4 SQL translation into XRA with TIDs

If we add an artificial primary key to all tuples, we *can* make correct translations for nested queries. The *tuple identifier* or *TID* is a unique number, relative to the TIDs of the other tuples in a specific relation.

4.1 Example translation

We will give a taste of the solution using TIDs in this subsection. The translation of the following problematic nested SQL query with grouping and with an aggregate function SUM will be discussed:

SQL query nr. 2:

```
SELECT *
FROM R
WHERE R.b >  (SELECT SUM(S.c)
              FROM S
              WHERE R.x = S.x);
```

Suppose the database contains the tables R and S as given here (note that R contains a duplicate tuple). The result of the SQL statement is shown as well:

R.x	*R.b*
1	40
1	40

S.x	*S.c*
1	10
1	20

$\Longrightarrow$

R.x	*R.b*
1	40
1	40

The SQL query result is calculated according to the nested-iteration method. For every tuple in the relation R, we calculate the subquery. The sum over the column S.c is 30 for each of the two original R-tuples.

To show how TIDs work, the correct translation into XRA with TIDs (see the `tid` column in the grouping construct) is:

Dupl. translation nr. 2tid:

```
?project((R.x, R.b),
        select(R.b > SUM(S.c),
               gb((R.tid, R.x, R.b), (SUM(S.c)),
                      select(R.x = S.x,
                             cp(R', S')))))
```

We show the tables R' and S', which are R and S each with an added *tid* column, as well as the temporary table after the "cp" and "select" operation, that is the last two lines. (S instead of S' gives a correct result as well. This is not so if the subquery in its turn contains a subquery again.)

R.x	*R.b*	*R.tid*
1	40	1
1	40	2

S.x	*S.c*	*S.tid*
1	10	1
1	20	2

R.x	*R.b*	*R.tid*	*S.x*	*S.c*	*S.tid*
1	40	1	1	10	1
1	40	1	1	20	2
1	40	2	1	10	1
1	40	2	1	20	2

When grouping is done on R.tid, then for every R-tuple the SUM over S.c is calculated separately. This is exactly what the SQL semantics prescribes. The TID serves to distinguish duplicates (for the SUM calculation), and to remove duplicates which were generated in the Cartesian product. The latter removal is done through the grouping columns in the "gb", which works just like "uniq(project(...))", except for aggregate function calculation.

If we did not group on R.tid, but only on R.x and R.b, then the XRA result would have been empty (no tuples), and therefore wrong. This was exactly the case gb-join in the previous section's analysis.

4.2 General XRA with TIDs translation

The general characteristics of the nested predicate translation are covered in this subsection.

We consider the evaluation of a nested predicate, with a subquery at level n. Let I', the inner tuples, contain the Cartesian product of this subquery's FROM relations, including their TIDs. Let O', the outer tuples, contain the Cartesian product of all its ancestor (level 1 through n-1) query blocks FROM relations plus TIDs. The TIDs in O' are called O-tids.

When we have TIDs at our disposal, we use either

- uniq(project((O-tids, O-attributes), join(O', I')))
- or, if we need to group:
  uniq(project((O-tids, O-attributes), gb((O-tids, OI-attributes), OI-aggregates, join(O', I')))).

to translate nested predicates in both WHERE and HAVING clauses.

Selection of tuples is not shown for simplicity, like in the previous section. The result now does satisfy the constraints for SQL results. Also notice the simplicity of the solution.

Why does the first option with uniq-project work?

First, the outer tuples O' are joined with the inner tuples I'. Then, O-tids and O-attributes (satisfying some search condition) are projected out. It is very likely that duplicates are among the resulting tuples from O'. They are both generated and original duplicates; generated duplicates were formed by the combination project-join. The O-tids form an artificial key for O'. The uniq operator therefore returns the original duplicates, and eliminates the generated duplicates. This is exactly what SQL requires.

An extra project is needed to remove the O-tids. Only the other columns are meaningful to the user and SQL. Those columns need not be all O-attributes, but can be some value expressions using O-attributes only. These value expressions come from the SELECT clause of the outermost query block. If the nested predicate was part of a more complex search condition or of a subquery, or if

another nested search condition (HAVING) is to be applied, then the TIDs are still needed.

Why does the second option with uniq-project-gb work?

The grouping operator gb can be caused by a GROUP BY clause in the subquery, and/or an aggregate function over either O or I tuples within a subquery. Per tuple in the outer block, grouping will be performed. Namely, we group on O-tids as well. If generated duplicates of O-tuples were present, the columns are *not* counted several times by the aggregates. This is as SQL requires. The uniq-project serves the same purpose as in the first solution, and an extra project may be needed as before.

The example in the previous subsection uses the second option. There, the grouping attributes are just O-attributes. Therefore, the extra uniq-project is not needed, and the project alone suffices.

Complex nested search conditions can be translated correctly too, now. The translation uses the TIDs in A' and B', the results of the partial nested search conditions scA and scB, respectively. For comparison, unnested search conditions "scA" are translated thus: select(scA, O').

"ScA AND scB" is translated into intersection(A', B'), or select(scA, B') if scA was not nested.

For "scA OR scB" we could use uniq(union(A', B')). This is likely to perform better than: union(diff(A', B'), B').

"NOT scA" becomes diff(O', A').

4.3 Syntax-directed translation of nested predicates

In this subsection, a general framework for the translation of nested predicates is given, to show how the TIDs are used in the complete translation. Not all details are covered. For the complete syntax-directed SQL to XRA translation, see [Verbr1].

4.3.1 Nested predicates

Nested predicates in SQL are all translated the same way, except the nested quantified predicate with ALL. For ALL, "$\forall$x Pred(x)" is equivalent with "$\neg\exists$x $\neg$Pred(x)". The latter form can be translated into XRA multi-set operations.

SQL syntax	XRA syntax	C
<nested predicate>	<XRA nested sc(subexpression)> ::=	
	uniq(project(	4
	allcolumns(subexpression),	3
	select(<XRA simple sc from nesting>,	2
	<XRA subquery(subexpression)>	1
	)))	

Numbered comments:

1. This is the subquery translation into XRA (see next translation). The subexpression tree was passed down by the translation process as a parameter. It contains the translation of the outer (sub)query: an XRA expression over O', containing the O-tids.
2. This is the search condition from the <nested predicate>, joining O' with I'. For example, for *SQL query nr. 2*, this condition is "R.b > SUM(S.c)", when we use column names.

   The expression "select(...)", now, has as a result columns from O' *and* from I'.
3. The function *allcolumns* returns all columns of O' including the O-tids.
4. The "uniq-project" is needed for extracting the O' columns and removing duplicates. Now the result contains only the O' columns satisfying the conditions.

4.3.2 Subqueries

SQL subqueries look like SQL queries, except that ORDER BY and a <select list> of length more than one are not allowed in subqueries.

SQL syntax	XRA syntax	C
<subquery>	<XRA subquery(subexpression)> ::=	1
SELECT [DISTINCT] <valexpr> FROM <table list> [WHERE <where sc>] [GROUP BY <grouping attrs>] [HAVING <having sc>]	<XRA having sc(gb((*allcolumns*(subexpression), <XRA grouping attrs>), *grouping_functions*, <XRA where sc(cp(subexpression, <XRA table list>))>))>	2 1

Most parts of the translation, except cp, are only needed if a corresponding part was present in the original SQL statement. Care should be taken for grouping: aggregate functions can be used in <valexpr>, or in the HAVING search condition (even in subqueries within it).

Numbered comments:

1. The "cp" is always needed. The subexpression contains the translation of the outer subquery. The translation of <table list>, a Cartesian product which forms I', is joined (or cp-ed) with the subexpression.

2. Also, we must group over the O' columns, including O-tids, from outer subqueries. Otherwise, the nested-iteration semantics are not simulated.

5 Conclusions and further research

Nested predicate translation into an extended relational algebra, e.g. XRA, is not feasible. Namely, XRA cannot distinguish between original duplicates, which must be kept, and generated duplicates, which must be eliminated. Correct duplicate handling appeared crucial to aggregate function calculation.

With an extra artificial primary key added to each tuple of a relation, called the tuple identifier, a syntax-directed translation can be made.

Further research includes the performance effects of tuple identifiers on the database system.

6 Acknowledgements

The database group at the CWI in Amsterdam, especially Arno Siebes and Martin Kersten, are thanked for their suggestions to improve this article.

References

[Astrahan] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, et al. (1976) System R: Relational Approach to Database Management. ACM Transactions on Database Systems 1(2), pp. 97-137.

[Ceri] Stefano Ceri, and Georg Gottlob. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. IEEE Trans. on Softw. Eng., Vol. SE-11, No.4, April 1985. pp. 324-345.

[Date] C.J. Date. An Introduction to Database Systems. Volume I. 3rd Edition. Addison-Wesley Publ. Company. 1981. Page 129.

[Date2] C.J. Date. Be careful with the SQL EXISTS function! In *Informatie*, December 1989, pp. 977-979.

[Hall] P.A.V. Hall. Optimization of a Single Relational Expression in a Relational Data Base System. IBM J. R. & D. **20**, No. 3 (1976).

[Kersten] M.L. Kersten, P.M.G. Apers, M.A.W. Houtsma, H.J.A. van Kuijk, and R.L.W. van de Weg. A Distributed, Main-Memory Database Machine. In Proc. of the 5th Int. Workshop on Database Machines, Karuizawa, Japan, Oct. 5-8, 1987; and in Database Machines and Knowledge Base Machines, M. Kitsuregawa, and H. Tanaka (eds.), Kluwer Academic Publishers, 1988, pp. 353-369.

[Kiessling] Werner Kiessling. On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates. Proceedings of VLDB 85, Stockholm. pp. 241-249.

[Kim] Won Kim. On Optimizing an SQL-like Nested Query, ACM Transactions on Database Systems, Vol.7, No. 3, September 1982, pp. 443-469.

[Murali] M. Muralikrishna. Optimization and Dataflow Algorithms for Nested Tree Queries. In Proc. 15th Int. Conf. on Very Large Data Bases, Amsterdam 1989, pp. 77-85.

[SQL] SQL standard, according to documents "Final Draft ISO 9075-1987(F) Database Language SQL", and "SQL Addendum-1 Error Log and corrected version".

[SQL2] "(ISO-ANSI working draft) Database Language SQL2 and SQL3", ANSI X3H2-89-110, February 1989.

[Ullman] Jeffrey D. Ullman. Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies. Computer Science Press, 1989. pp. 633-733.

[Verbrl] N.Th. Verbrugge. Translation of Nested SQL Queries into eXtended Relational Algebra. PRISMA document nr. 501, 1989.

[Wilschut] Annita Wilschut, Paul Grefen. PRISMA/DB1 XRA Definition. PRISMA document nr. 465. Sept. 1989.