

# Unnesting Scalar SQL Queries in the Presence of Disjunction

Matthias Brantner\*      Norman May      Guido Moerkotte  
Database Research Group  
University of Mannheim, Mannheim, Germany  
msb|norman|moer@db.informatik.uni-mannheim.de

## Abstract

*Optimizing nested queries is an intricate problem. It becomes even harder if in a nested query the linking predicate or the correlation predicate occurs disjunctively. We present the first unnesting strategy that can effectively deal with such queries.*

*The starting point of our approach is to translate SQL into the relational algebra extended by bypass operators. Then we present for the first time unnesting equivalences which are valid for algebraic expressions containing bypass operators. Applying these to the translated queries results in our effective unnesting strategy for nested SQL queries with disjunction. With an extensive experimental study (including three commercial DBMSs), we demonstrate the possible performance gains of our approach.*

## 1 Introduction

Nested queries easily become a performance bottleneck because in many cases, they demand a nested-loop evaluation. For conjunctive predicates this problem has been addressed successfully, e.g. [19, 25]. However, current unnesting techniques fail in the presence of disjunctive predicates. Despite the fact that disjunctions occurring inside nested queries are common in practice, we are not aware of any publication which treats unnesting nested queries which contain disjunctions, i.e. the linking or correlation predicate occur in a disjunction. For example, when asked about disjunctions in connection with nested queries, César A. Galindo-Legaria from the Microsoft SQL Server Group said: "We are running into it quite often."

**Key Idea.** Let us consider an example for an analytical query. Assume we are interested in all European suppliers that deliver a certain part with minimum supply costs or

have a minimal amount of this part available on stock. In SQL, this query can be formulated as follows:

```
SELECT s_acctbal, s_name, n_name, p_partkey,
       p_mfgr, s_address, s_phone, s_comment
FROM   part, supplier, partsupp, nation, region
WHERE  p_partkey = ps_partkey
AND    s_suppkey = ps_suppkey AND p_size = 15
AND    p_type LIKE '%BRASS'
AND    s_n_key = n_n_key AND n_r_key = r_r_key
AND    r_name = 'EUROPE'
AND    (ps_supplycost=(SELECT min(ps_supplycost)
                       FROM   partsupp, supplier,
                              nation, region
                       WHERE  s_suppkey = ps_suppkey
                              AND  p_partkey = ps_partkey
                              AND  s_n_key = n_n_key
                              AND  n_r_key = r_r_key
                              AND  r_name = 'EUROPE')
OR ps_availqty > 2000)
ORDER BY s_acctbal desc, n_name, s_name, p_partkey
```

This query is very similar to TPC-H Query 2. Hence, we refer to it as Query 2d. It exhibits two key components: (1) it features a nested, correlated subquery, and (2) it contains a disjunction. Our unnesting strategy is capable of optimizing nested queries whose linking or correlation predicates occur disjunctively. The key idea is that the nested query block needs to be evaluated only for those tuples of the outer query block that do not pass the cheap and simple predicate `ps_availqty > 2000`. For those tuples, we are currently restricted to an inefficient nested-loop evaluation. However, our novel unnesting technique allows to employ more efficient evaluation algorithms. Consequently, our approach exploits both the short-cut evaluation of the disjunction and the power of unnesting nested queries.

**Our Approach.** The starting point of our approach is to translate SQL into the relational algebra extended with bypass operators [17]. Then, we apply our novel unnesting equivalences, which can cope with disjunctions on a large variety of nested queries. As a result, nested query blocks are removed, and the resulting queries can be evaluated much more efficiently.

\*This work was supported by the Deutsche Forschungsgemeinschaft under grant MO 507/10-1.

Applying unnesting at the algebraic level has mainly three advantages. (1) It is possible to give rigorous correctness proofs for the unnesting equivalences. (2) Unnesting techniques stated as algebraic equivalences are query language independent as long as the query language is translatable into the algebra. (3) Unnesting equivalences can be used during plan generation. This allows to apply them in a cost-based manner. The latter is especially important in our case, since some unnesting strategies do not always result in better plans.

**Contributions.** The main contributions of our paper are:

- We present equivalences for unnesting algebraic expressions with bypass operators to handle disjunctive linking and correlation predicates where the predicate involves  $\theta \in \{=, \neq, <, \leq, >, \geq\}$ .
- We show how they can be used to effectively unnest SQL queries with scalar subqueries featuring an arbitrary aggregation function in the `select` clause.
- Our techniques can be applied not only to queries with exactly one nested block (simple queries), but also to queries whose nesting has a linear or even a tree structure.
- We provide experimental results demonstrating the performance improvements that can be achieved by our approach.

Several additional details of our technique can be found in the technical report [2]:

- In this paper, we restrict our exposition to nested queries in the `where` clause. But the generalization to nesting in the `select` clause is straightforward.
- We present how our approach can be applied to quantified table subqueries with the operators EXISTS, NOT EXISTS, IN, and NOT IN.

**Limitations.** As a current limitation, we restrict ourselves to queries exhibiting direct correlation: the correlation predicate only refers to attributes of the current block and the direct outer block.

**Structure of this Paper.** The remainder of the paper is organized as follows. Section 2 briefly introduces preliminaries. Section 3 contains our unnesting techniques for scalar subqueries. After introducing these approaches, we show their effectiveness with an experimental study (Section 4). At the end, we summarize related work in Section 5 and conclude the paper with Section 6.

## 2 Preliminaries

### 2.1 Terminology

A *query block* is a `select-from-where` expression. A query containing a query block nested in another query block is called a *nested query*. The containing query block is called *outer query block*, and the contained block is called *inner query block*. An inner query block is also called *nested query block*. Let  $p$  be a predicate occurring in the `where` clause of an inner query block. If  $p$  refers to attributes defined in the outer query block and to attributes defined in the inner query block,  $p$  is called a *correlation predicate*, and the inner query block is called *correlated*. A predicate  $q$  in the `where` clause of the outer query block which contains the inner query block as an argument is called *connection predicate*. The operator used in the connection predicate is called *connection operator*. Connection predicates are also called *linking predicates* [4].

If a linking predicate occurs in a disjunction as, for example, in the introductory query, this is called *disjunctive linking*. Analogously, if the correlation predicate occurs in a disjunction, this is called *disjunctive correlation*.

### 2.2 Classification

Kim introduced four types of nested query blocks [19]:  $A$ ,  $N$ ,  $JA$ , and  $J$ . Let us refer to a nested query block as  $B$ . If  $B$  contains an aggregate function,  $B$  is of type  $A$  or  $JA$  and is called *scalar subquery*.  $B$  must return a single column. If  $B$  contains a correlation predicate, it is of type  $J$  or  $JA$ . A nested query block that neither has an aggregate function nor a correlation predicate is of type  $N$ . Query blocks with an aggregation function but no correlation predicate are of type  $A$ . Nested query blocks of type  $N$  or  $J$  are called *table subqueries*. They are connected to their outer query block using the *positive linking operators* EXISTS, SOME/ANY, and IN or *negative linking operators* NOT EXISTS, ALL, and NOT IN. We cover these cases in our technical report [2] only.

While Kim concentrates on classifying single nested query blocks, Muralikrishna classifies queries according to the nesting structure [22]. He subdivides queries with more than one nested block into linear and tree queries: A *Linear (Nested) Query* is a query where at most one block is nested within any block. A *Tree (Nested) Query* is a query with at least one block which has two or more blocks nested within it at the same level. We complete this classification and call a query with exactly one nested block a *Simple (Nested) Query*.

## 2.3 Algebra

The domain of the relational algebra consists of sets of tuples. The core algebra contains the following operators: union ( $\cup$ ), intersection ( $\cap$ ), set-difference ( $\setminus$ ), projection ( $\Pi$ ), renaming operator ( $\rho$ ), and selection ( $\sigma$ ) [13]. We denote disjoint union by  $\dot{\cup}$ .

For the purpose of this paper, we extend the core algebra by five operators: a unary grouping operator ( $\Gamma$ ), a binary grouping operator ( $\Gamma$ ) [5, 28], a leftouterjoin ( $\bowtie^{g:f(\emptyset)}$ ) [5, 7], a numbering operator ( $\nu$ ), and a map operator ( $\chi$ ). Implementations for binary grouping operators ( $\Gamma$ ) can be found in [21]. Given the definition of the binary grouping operator in Fig. 1, we define the unary grouping operator. The leftouterjoin ( $\bowtie^{g:f(\emptyset)}$ ) is required to address the ‘‘count bug’’ [18], i.e. losing a tuple due to an empty group. Therefore, the function  $f$  assigns meaningful values to empty groups. The numbering operator ( $\nu$ ) characterizes each tuple with a unique deterministic number (e.g. a physical tuple identifier). We mainly use the map operator to apply a function to each tuple. Both are required for the unnesting techniques introduced in Section 3.

Figure 1 summarizes the formal definition of the five operators. As a final important extension of our algebra, we allow subscripts to contain algebraic expressions. In our case, such subscripts result from translating nested query blocks in the *where* clause, i.e. algebraic operators appear in selection predicates.

In order to effectively deal with disjunction, we need algebraic operators that split their output into two streams: a positive and a negative one. For example, a selection produces a *positive stream* containing all those tuples for which the selection predicate evaluates to true; the *negative stream* contains the remaining tuples. These operators are called *bypass operators* [17]. To denote the positive and negative streams of a bypass operator, we use the superscripts  $+$  and  $-$ , respectively.

For this paper, we need a bypass selection ( $\sigma^\pm$ ), and a bypass join ( $\bowtie^\pm$ ). Their definitions are given in Figure 1. Although the algebra is based on sets of tuples, our approach is also applicable for an algebra on bags. However, the focus of this paper is on sets. Section 3.7 and proofs in the appendix of our technical report [2] elaborate on the applicability of our techniques on bags.

## 3 Unnesting Scalar Subqueries

Unnesting scalar queries is difficult and error-prone. Particularly, empty groups and duplicates [18] have been sources of errors. As a new challenge, we now support disjunctive linking and correlation.

This section is organized as follows. First, we discuss the basic idea of our approach by means of two simple

Extended operators:

$$\begin{aligned}
e_1 \Gamma_{g;A_1 \theta A_2;f} e_2 &:= \{x.A_1 \circ [g : G] \mid x \in e_1 \wedge \\
&\quad G = f(\{y \mid y \in e_2 \wedge x.A_1 \theta y.A_2\})\} \\
\Gamma_{g;=A;f}(e_1) &:= \Pi_{A:A'}(\Pi_{A':A}(e_1) \Gamma_{g;A=A';f}(e_1)) \\
e_1 \bowtie_p^{g:f(\emptyset)} e_2 &:= e_1 \bowtie_p e_2 \cup \{x \circ z \mid x \in e_1 \wedge \\
&\quad \exists y \in e_2 : p(x, y) \wedge \mathcal{A}(z) = \mathcal{A}(e_2) \wedge \\
&\quad g \in \mathcal{A}(e_2) \wedge \forall a \in (\mathcal{A}(e_2) \setminus g) : \\
&\quad (z.a : \text{NULL} \wedge z.g : f(\emptyset))\} \\
\nu_A(e) &:= \{t_i \circ [A : i] \mid e = \{t_1, \dots, t_n\}\} \\
\chi_{a:e_2}(e_1) &:= \{x \circ [a : e_2(x)] \mid x \in e_1\}
\end{aligned}$$

Bypass operators:

$$\begin{aligned}
\sigma_p^+(e) &:= \{x \mid x \in e \wedge p(x)\} \\
\sigma_p^-(e) &:= e \setminus \sigma_p^+(e) \stackrel{*}{=} \{x \mid x \in e \wedge \neg p(x)\} \\
e_1 \bowtie_p^+ e_2 &:= \{x \circ y \mid x \in e_1 \wedge y \in e_2 \wedge p(x, y)\} \\
e_1 \bowtie_p^- e_2 &:= (e_1 \times e_2) \setminus (e_1 \bowtie_p^+ e_2) \\
&\stackrel{*}{=} \{x \circ y \mid x \in e_1 \wedge y \in e_2 \wedge \neg p(x, y)\}
\end{aligned}$$

\*: only valid for two-valued logic (cf. [17] for details).  $[\cdot]$  denotes tuple construction.  $\circ$  denotes tuple concatenation.  $\mathcal{A}(R)$  is the set of attributes of relation  $R$ .  $\mathcal{F}(e)$  is the set of free attributes that occur freely in expression  $e$ .

Figure 1. Operators of the algebra

queries. Second, we present the general solution in the form of unnesting equivalences. On their left-hand side, they have a selection whose predicate contains disjunctively a nested algebraic expression with a top-level aggregation. On their right-hand side, they introduce a bypass operator. After the discussion of our equivalences, we move on to more advanced issues, i.e. unnesting of linear queries, tree queries, and a discussion of duplicate handling.

Scalar subqueries of type  $JA$  are easy to handle. Their result can be computed independently of the outer query, and materialization costs are negligible. Thus, it suffices to materialize the computed result. As their treatment is so simple, we do not discuss them any further but concentrate on the more challenging type  $JA$  queries.

### 3.1 Disjunctive Linking

In the following query, the subquery is of type  $JA$ , as it contains a predicate which refers to the attribute  $A_2$  defined in the outer block and the attribute  $B_2$  defined in the inner block:

```

SELECT DISTINCT *
FROM R
WHERE A1 = (SELECT COUNT(DISTINCT *)
             FROM S
             WHERE A2 = B2)
OR A4 > 1500

```

Q1

The linking predicate compares the attribute  $A_1$  with the result of the aggregation (i.e. count) in the inner query’s

`select` clause. Moreover, the linking predicate occurs in a disjunction. Translation into the algebra yields the following expression:

$$\sigma_{A_1=\text{count}(\sigma_{A_2=B_2}(S)) \vee A_4 > 1500}(R).$$

Fig. 2(a) presents this canonical evaluation plan in a more readable form.

For the evaluation of this query, the inner query has to be evaluated for every tuple produced by the outer query block, i.e. in nested loops. Obviously, this is not very efficient. In order to unnest type *JA* queries in the conjunctive case, it is common practice to apply grouping on the correlation attributes of the inner query to perform the aggregation. Then, an outerjoin is performed to accomplish the match with the tuples from the outer query block with the grouped and aggregated result. The following equivalence captures this procedure:

$$\begin{aligned} & \sigma_{A_1 \theta f(\sigma_{A_2=B_2}(S))}(R) \\ \equiv & \Pi_{\mathcal{A}(R)}(\sigma_{A_1 \theta g}(\mathbb{R} \bowtie_{A_2=B_2}^{g:f(\emptyset)} (\Gamma_{g:=B_2;f}(S)))). \end{aligned} \quad (1)$$

If the predicate in the outer query block was a conjunction, we could apply this equivalence without hesitation. However, if we apply this equivalence to the translation of the query, the resulting plan contains an outerjoin with a disjunctive join predicate. In this case, the only known implementation is the rather inefficient nested-loop implementation.

Let us take a closer look at the query. Assume that a tuple from *R* satisfies  $A_4 > 1500$ . Then we do not have to check  $A_1 = \dots$  for it: it qualifies independently of the result of this check. Further, if a tuple from *R* does not satisfy  $A_4 > 1500$ , it must satisfy  $A_1 = \dots$  in order to qualify. Thus, it does make sense to split the tuple stream produced by scanning *R* into two independent streams: one containing those tuples satisfying  $A_4 > 1500$  and one with the remaining tuples. The latter then needs to be filtered by  $A_1 = \dots$ . Finally, as the two streams are disjoint, a disjoint union ( $\dot{\cup}$ ) on them suffices to produce the final result. Bypass operators capture exactly this kind of reasoning. This is why we want to use them for unnesting. Let us therefore introduce a bypass selection with predicate  $A_4 > 1500$ . The following algebraic expression results from this:

$$\begin{aligned} e &= e_1 \dot{\cup} e_2 \\ e_1 &= \sigma_{A_4 > 1500}^+(R) \\ e_2 &= \sigma_{A_1=\text{count}(\sigma_{A_2=B_2}(S))}(\sigma_{A_4 > 1500}^-(R)). \end{aligned}$$

Fig. 2(b) shows the more readable result. The positive stream of the bypass selection (denoted by a solid line) directly contributes to the final result. In addition, the negative stream (denoted by dots) is filtered by a selection with the algebraic equivalent of  $A_1 = \dots$ .

With this expression as a starting point, we can derive the unnested bypass plan shown in Fig. 2(c). Those tuples of *R* that satisfy the predicate  $A_4 > 1500$  directly contribute to the result. Only for the remaining tuples, we need to check the condition expressed by  $A_1 = \dots$ . This check is represented in the plan by the same trick used to unnest conjunctively nested queries. In a first step, we group by the linking attribute  $B_2$  of the inner query and calculate the aggregate. Then, we perform an outerjoin. For those tuples of *R* that do not find a join partner, the default handling of the outerjoin assures correctness. Last, we evaluate the linking predicate. It has been rewritten since the aggregation result has been materialized in the attribute *g*. A final projection on the attributes of *R* guarantees the same schema in the positive as well as the negative stream before unioning the two streams.

**Remark.** It is important to recognize that commuting the bypass selection with the selection in the negative stream (see Fig. 2(d)) is also feasible. This enables further optimization potential. Assume that the second predicate is expensive to evaluate. Then it may be cheaper to perform the selection first. This situation is recognized by comparing ranks of the predicates: the one with the lower rank should be evaluated first [26]. For a predicate *p*, the rank ( $\text{rank}(p)$ ) is defined as  $\frac{s-1}{c}$ , where *s* is the selectivity of predicate *p* and *c* is the cost required to evaluate *p*.

That is, if the predicate  $A_4 > 1500$  was a very expensive one, we could evaluate the subquery first. In this case, the selection checking the linking predicate turns into a bypass selection, and the predicate  $A_4 > 1500$  is evaluated only in the bypass selection's negative stream. A projection on the attributes of *R* in both streams ensures the final schema.

### 3.2 Disjunctive Correlation

Not only the linking predicate can occur in a disjunction. The following query contains a disjunctively occurring correlation predicate, i.e. disjunctive correlation:

```
SELECT DISTINCT *
FROM R
WHERE A1 = (SELECT COUNT(*)
            FROM S
            WHERE A2 = B2
            OR B4 > 1500) Q2
```

The aggregation function in the `select` clause of the nested query combines all tuples that pass the correlation predicate  $A_2 = B_2$  or the simple predicate  $B_4 > 1500$ .

Similar to the canonical translation of Query Q1, but with the disjunction in the selection predicate of the nested selection, the canonical translation gives us (see also Figure 3(a))

$$\sigma_{A_1=\text{count}(\sigma_{A_2=B_2 \vee B_4 > 1500}(S))}(R).$$

Unnesting is not possible with any of the existing techniques. For the following, we refer to the plan in Fig. 3(b).

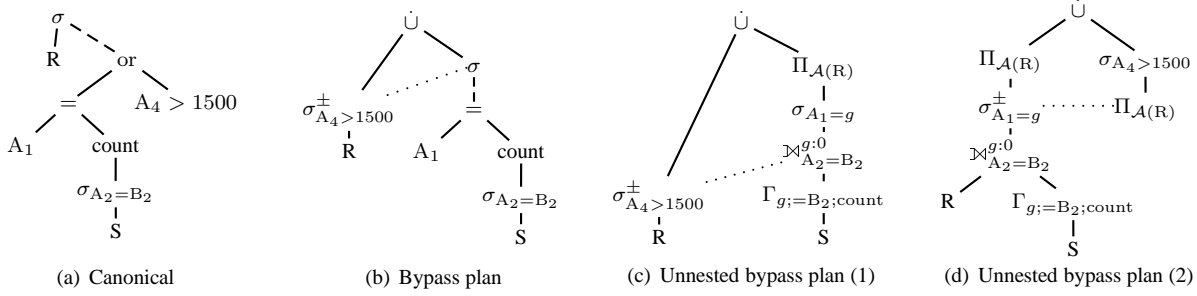


Figure 2. Unnesting strategy for Q1 (sketch)

The general idea to unnest this query is based on two facts: (1) the aggregation function (count) is decomposable [6], and (2) the predicate  $B_4 > 1500$  can be evaluated independently of the outer query. This allows us to calculate the total count of the inner query from adding up the counts calculated for two disjoint subsets. Take a look at the bottom of the plan in Fig. 3(b). In the positive stream of the bypass selection (denoted by a solid line), we count all tuples from relation  $S$  that satisfy the predicate  $B_4 > 1500$ . Those tuples of  $S$  that do not satisfy  $B_4 > 1500$  go into the negative stream. Here, they have to pass the correlation predicate before they contribute to the total count. Hence, we group them and evaluate the count function for each group. Analogously to the general unnesting strategy (see Eqv. 1), we apply an outerjoin to perform the match with the outer relation  $R$  and — in order to avoid the count bug — assign 0 to the attribute  $g_1$  for those tuples from  $R$  that do not have a join partner. At the end, we need a map to add up the separately calculated values for  $g_1$  and  $g_2$  to give the total count  $g$ . The subsequent selection with predicate  $A_1 = g$  checks the linking predicate. The final projection assures that the result only contains attributes from  $R$ .

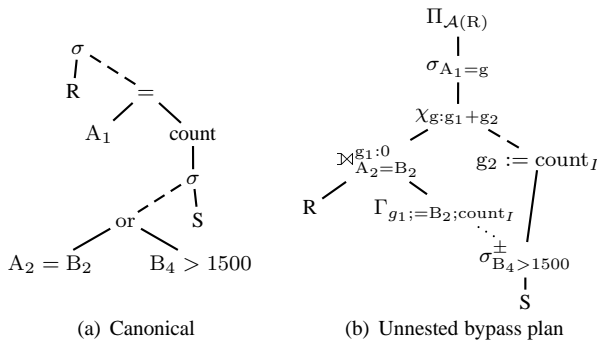


Figure 3. Unnesting strategy for Q2 (sketch)

### 3.3 Equivalences

Before we can present our unnesting rewrites for scalar queries of type  $JA$ , we need to define *decomposability* of

aggregate functions [6]. Let  $X, Y$ , and  $Z$  be sets with  $X = Y \cup Z$  and  $Y \cap Z = \emptyset$ . A scalar aggregate function  $f : X \rightarrow \mathcal{N}$  is *decomposable* if there exist functions

$$\begin{aligned} f_I : X &\rightarrow \mathcal{N}' \\ f_O : \mathcal{N}', \mathcal{N}' &\rightarrow \mathcal{N} \end{aligned}$$

with  $f(X) = f_O(f_I(Y), f_I(Z))$ .

The SQL aggregation functions used most often are decomposable:

$$\begin{aligned} \text{count}(X) &\equiv \text{count}_I(Y) + \text{count}_I(Z) \\ \text{sum}(X) &\equiv \text{sum}_I(Y) + \text{sum}_I(Z) \\ \text{avg}(X) &\equiv \frac{\text{sum}_I(Y) + \text{sum}_I(Z)}{\text{count}_I(Y) + \text{count}_I(Z)} \\ \text{min}(X) &\equiv \text{min}_O(\text{min}_I(Y), \text{min}_I(Z)) \\ \text{max}(X) &\equiv \text{max}_O(\text{max}_I(Y), \text{max}_I(Z)). \end{aligned}$$

The discussion of our equivalences (see Fig. 4) is split into two parts. In the first part, we discuss unnesting equivalences for queries with disjunctive linking. In the second part, we advance to unnesting equivalences for queries with disjunctive correlation.

#### 3.3.1 Disjunctive Linking

In the equivalences, let  $f$  be an aggregation function.

**Equivalences 2 and 3** are used to unnest scalar queries whose linking predicate occurs disjunctively. The former postpones the evaluation of the unnested subquery into the negative stream of a bypass selection. Basically, the unnesting technique is adapted from Eqv. 1. The idea of this equivalence has already been explained using Query Q1. Fig. 2(c) depicts this strategy.

The latter equivalence is used for first evaluating the unnested subquery, i.e. the linking predicate, and postpone the evaluation of the second predicate into the negative stream of the bypass selection. Fig. 2(d) visualizes this strategy.

$$\begin{aligned} \sigma_{p \vee A_1 \theta(f(\sigma_{A_2=B_2}(S)))}(\mathbf{R}) &\equiv e_1 \dot{\cup} e_2 \\ e_1 &:= \sigma_p^+(\mathbf{R}) \end{aligned} \quad (2)$$

$$\begin{aligned} e_2 &:= \Pi_{\mathcal{A}(R)}(\sigma_{g\theta A_1}((\sigma_p^-(\mathbf{R})) \bowtie_{A_2=B_2}^{g:f(\theta)} (\Gamma_{g:=B_2;f}(S)))) \\ \sigma_{p \vee A_1 \theta(f(\sigma_{A_2=B_2}(S)))}(\mathbf{R}) &\equiv \Pi_{\mathcal{A}(R)}(e_1 \dot{\cup} e_2) \end{aligned} \quad (3)$$

$$\begin{aligned} e_1 &:= \sigma_{g\theta A_1}^+(\mathbf{R}) \bowtie_{A_2=B_2}^{g:f(\theta)} (\Gamma_{g:=B_2;f}(S)) \\ e_2 &:= \sigma_p(\sigma_{g\theta A_1}^-(\mathbf{R}) \bowtie_{A_2=B_2}^{g:f(\theta)} (\Gamma_{g:=B_2;f}(S))) \end{aligned} \quad (4)$$

$$\begin{aligned} \sigma_{A_1 \theta f(\sigma_{A_2=B_2 \vee p}(S))}(\mathbf{R}) &\equiv \Pi_{\mathcal{A}(R)}(\sigma_{A_1 \theta g}(\chi_{g:f_O(g_1, e_2)}(e_1))) \\ e_1 &:= R \bowtie_{A_2=B_2}^{g_1:f_I(\theta)} (\Gamma_{g_1:=B_2;f_I}(\sigma_p^-(S))) \\ e_2 &:= f_I(\sigma_p^+(S)) \end{aligned} \quad (4)$$

$$\begin{aligned} \sigma_{A_1 \theta_1 f(\sigma_{A_2 \theta_2 B_2 \vee p}(S))}(\mathbf{R}) &\equiv \Pi_{\mathcal{A}(R)}(\sigma_{A_1 \theta_1 g}((R') \Gamma_{g;t_1=t_1';f}(\rho_{t_1' \leftarrow t_1}(e_1 \dot{\cup} e_2)))) \\ R' &:= \nu_{t_1}(R) \\ e_1 &:= R' \bowtie_{A_2 \theta_2 B_2}^+ S \\ e_2 &:= \sigma_p(R' \bowtie_{A_2 \theta_2 B_2}^- S) \end{aligned} \quad (5)$$

**Figure 4. Equivalences for disjunctive queries of type  $JA$**

### 3.3.2 Disjunctive Correlation

**Equivalence 4** handles queries whose correlation predicate occurs in a disjunction. Its limitation is that the predicate expression  $p$  must not be a subquery itself. Moreover, this equivalence requires the aggregation function to be decomposable and the correlation predicate to be an equality predicate<sup>1</sup>. Fig. 3 illustrates the idea of this equivalence for the query from Section 3.2. Its main idea is to generate partial, intermediate results, which are then combined by the subsequent map operator.

**Equivalence 5** in contrast, is more generally applicable. There are no restrictions on the aggregate function. In addition, predicate  $p$  may contain a nested query. The bypass join generates one positive stream for those tuples satisfying the correlation predicate and a complementary negative one where  $p$  is checked. Beforehand, we need to introduce a numbering operator  $\nu$ , which enables us to correctly reassemble the results during the binary grouping.

### 3.4 Completeness of Equivalences

Our equivalences handle all cases of scalar subqueries with disjunctive linking and correlation. Thereby, the link-

<sup>1</sup>Note that the DISTINCT versions of the aggregation functions COUNT, SUM, and AVG are not decomposable. In this case, Eqv. 5 must be applied.

ing predicate can consist of an arbitrary linking operator ( $\{=, \neq, <, \leq, >, \geq\}$ ).

Let us make sure that the canonical translation of a scalar subquery always leads to a pattern that matches the left-hand side of one of our equivalences. In this situation, the canonical translation results in an aggregate function call  $f$  as top-level member of a selection predicate, which is part of the linking predicate. In Eqv. 2 and 3, this corresponds to disjunctive linking. The argument of the aggregation function is again a selection checking for the correlation predicate, which in Eqv. 4 and 5 occurs in a disjunction. Remember that the former is a special case of the latter, where  $p$  must not be a subquery itself and the aggregation function  $f$  must be decomposable.

### 3.5 Tree Queries

Tree queries of type  $JA$  can be unnested quite easily by successive applications of our equivalences. Consider the following query:

```
SELECT DISTINCT *
FROM R
WHERE A1 = (SELECT COUNT(DISTINCT *)
            FROM S
            WHERE A2 = B2)
OR
A3 = (SELECT COUNT(DISTINCT *)
      FROM T
      WHERE A4 = C2)
```

**Q3**

Figure 5 illustrates the canonical translation and the result of the following two steps. In a first step, we unnest the query using Eqv. 2 applied to the predicate, i.e. the subquery, having the lowest rank. In the second step, we have to make a choice: either we apply Eqv. 2 again, if there exists another subquery on the same level, or we apply Eqv. 1, if this is not the case. Because none of the subqueries contains a nested query, we apply Eqv. 1.

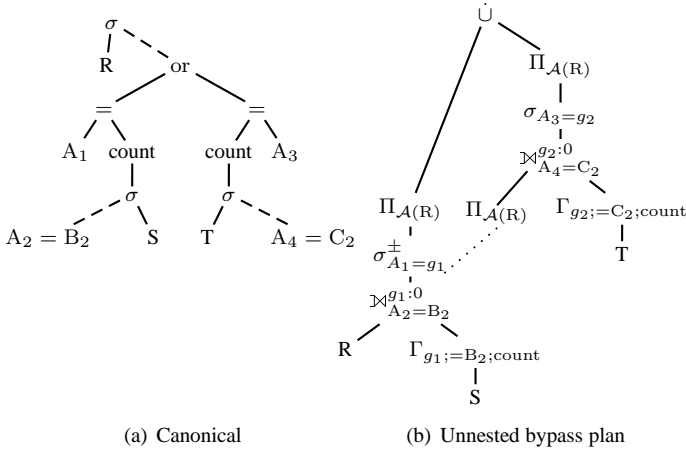


Figure 5. Unnesting strategy for Q3 (sketch)

### 3.6 Linear Queries

Linear JA queries are a special case of disjunctive correlation. The second predicate in the disjunction is again a linking predicate, as in the following query:

```

SELECT DISTINCT *
FROM R
WHERE A1 = (SELECT COUNT(DISTINCT *)
            FROM S
            WHERE A2 = B2
            OR
            B3 = (SELECT COUNT(DISTINCT *)
                FROM T
                WHERE B4 = C2))

```

**Q4**

As you can see in Fig. 6, we start with the canonical translation (see Fig. 6(a)) and unnest in a top-down fashion. In a first step, we apply Eqv. 5. The result is shown in Fig. 6(b). From here, it becomes obvious that for the deepest nested expression Eqv. 1 can be applied which yields the final plan shown in Fig. 6(c).

### 3.7 Duplicate Handling

Let us make sure that all equivalences mentioned in this section are also correct when they are based on an algebra over multisets. The right-hand side of Equivalences 2, 3,

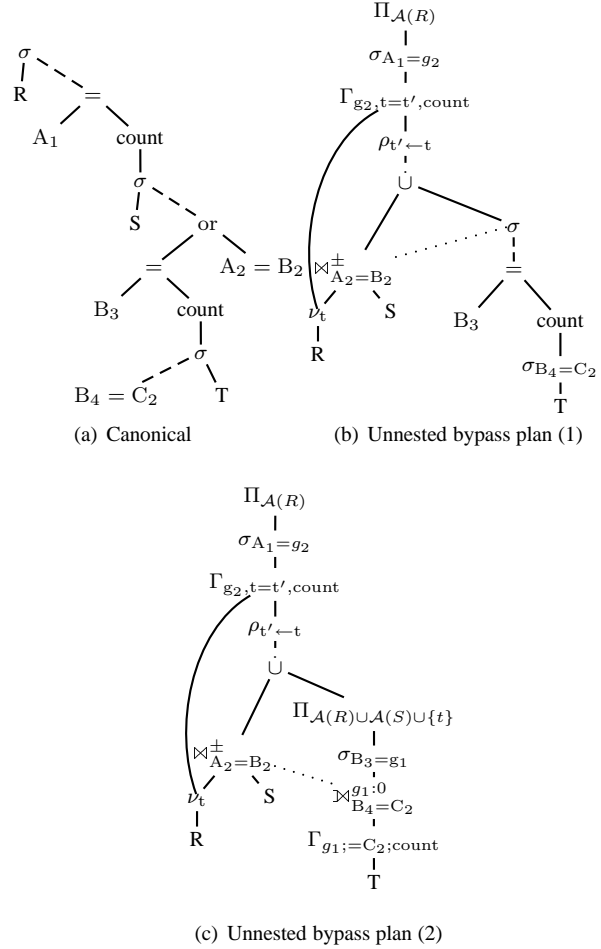


Figure 6. Unnesting strategy for Q4 (sketch)

and 4 contains a unary grouping on the nested query block's input, followed by a leftouterjoin. We observe that after grouping on the correlation attribute of the inner query, each value of the grouping attributes occurs exactly once. This key is later used as join attribute in the leftouterjoin. As a result, this join either finds exactly one matching tuple for each tuple resulting from the outer query block, or it keeps the outer block's tuple in order to preserve empty groups. Hence, the cardinality of the leftouterjoin is exactly the one of the outer relation R.

In Eqv. 4, we have already ensured correctness of the duplicate semantic for expression  $e_1$  above. The map operator does not influence duplicates, as it only computes the correct aggregate value.

The numbering operator  $\nu$  in Eqv. 5 turns the multiset R into a set and thereby ensures correctness for multisets.

Each equivalence introduces one bypass operator. This operator, in the unnested plan, partitions its input into two disjoint sets. Thus, it neither creates duplicates nor discards

any tuples. Moreover, the final union is defined for multisets, ensuring that duplicates are handled correctly.

## 4 Evaluation

To demonstrate the effectiveness of our unnesting techniques, we performed an extensive evaluation of them. Specifically, we measured the execution times of the canonical and the unnested plans using our hybrid relational and XML DBMS Natix [10]. Additionally, we compared the resulting evaluation times with those measured for three major commercial database management systems, for anonymity reasons henceforth nicknamed S 1, S 2, and S 3. For the same reason, we cannot present specific query evaluation plans for the commercial systems. However, the strategy can be predicted by comparing these evaluation times with those resulting from the execution of the canonical plans in Natix.

In this paper, we focus on the evaluation of simple queries. For queries with a more complex nesting structure, i.e. linear and tree queries, the performance gains observed for simple queries exponentiate. We refer the reader to our technical report [2] for the results of these experiments.

After a brief description of the experimental setup, we present two evaluations for simple scalar subqueries: (1) scalar subqueries with disjunctive linking and (2) scalar subqueries with disjunctive correlation.

### 4.1 Datasets

The evaluation is performed on several data sets based on two schemas: (1) the schema of the TPC-H benchmark [29] and (2) the schema RST used for the example queries from Section 3. The latter schema contains three tables (R, S, and T), each consisting of four columns  $A_i \in \mathcal{A}(R)$ ,  $B_i \in \mathcal{A}(S)$ , and  $C_i \in \mathcal{A}(T)$  for  $i = 1 \dots 4$ .

The data sets for the TPC-H benchmark are generated using the benchmark generator (dbgen) with scaling factors ( $SF$ ) 0.01, 0.05, 0.5, 1, 5 and 10. This results in moderate database sizes of 11MB - 11GB.

For the independently scaled relations of the RST schema, we generated instances with scaling factors ( $SF$ ) 1, 5, and 10. This led to 10.000, 50.000, and 100.000 rows and amounts to small tables of 178K, 1.1M, and 2.1M. In the evaluations below, SF1 denotes the scaling factor of the outer query block and SF2 the scaling factor of the inner query block. We did not use larger scaling factors because neither the canonical plans nor the commercial systems scaled well.

## 4.2 Settings

We used two comparable PCs with 1 GB of RAM each for the experiments. Not all commercial systems are available for the same operating system. We executed Natix and two of the commercial systems on one of the PCs running Linux 2.6.11. The other commercial system ran on the other PC under Windows XP. All queries were executed with a cold buffer. Further, for optimizing the queries using the commercial systems, we used the highest optimization level possible. However, we did not create any indexes.

Because of the necessity to use two different systems, the resulting evaluation times are not exactly comparable. However, the growth of the resulting evaluation times already demonstrates the effectiveness of our unnesting approaches.

## 4.3 Results

Fig. 7 presents the results of our performance study. We aborted the execution of queries that took longer than six hours. These cases are denoted by n/a. The remaining execution times are shown in seconds.

### 4.3.1 Disjunctive Linking

We selected Query Q1 and, based on the TPC-H schema, the introductory Query 2d to evaluate simple queries with disjunctive linking and scalar subqueries.

For both queries, we executed two query execution plans in Natix. The first plan implements a canonical translation, the second results from the application of Eqv. 2. Figures 2(a) and 2(c) illustrate these strategies for Query Q1. The plans for Query 2d use the same strategies. The detailed plans are presented in our technical report [2].

Fig. 7(a) and 7(b) present the results for these queries. We observe that our unnested approach excels all other approaches — for the RST as well as the TPC-H data set. In comparison with our canonical approach, the performance numbers of the commercial systems for the RST data set allow to deduce that these systems execute the nested query in a nested-loop like fashion. Only the commercial system S 2 almost keeps up with our unnested approach. However, for the TPC-H data set our unnested approach even outperforms this system by one order of magnitude. The remaining commercial systems are surpassed by three to four orders of magnitude for the cases that finished within six hours.

### 4.3.2 Disjunctive Correlation

Besides the evaluation of JA queries with disjunctive linking, we performed an evaluation for queries with disjunctive correlation. Therefore, we executed Query Q2 using the

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	10.6	55.7	111	49.4	259	520	98.3	515	1029
S 2	0.19	0.33	0.52	0.92	1.17	1.30	1.95	2.13	2.52
S 3	5.06	25.1	50.1	25.7	144	267	49.8	259	558
<b>Natix</b>									
• canonical	10.9	54.9	109	46.8	235	474	88.5	450	899
• unnested	0.2	0.24	0.3	0.78	0.87	0.98	1.6	1.65	1.74

(a) Q1

	TPC-H Scaling Factor (SF)					
	0.01	0.05	0.5	1	5	10
System						
S 1	0.14	0.36	52.5	123	n/a	n/a
S 2	0.10	2.00	29.0	67.0	328	766
S 3	0.27	0.57	48.7	234	n/a	n/a
<b>Natix</b>						
• canonical	79.7	3631	n/a	n/a	n/a	n/a
• unnested	0.14	0.19	0.82	1.49	23.1	49.5

(b) Query 2d

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	16.7	90.3	184	82.7	445	905	165	892	1803
S 2	8.55	46.3	95.5	42.9	235	479	85.7	466	971
S 3	11.6	59.7	120	71.4	378	737	143	753	1519
<b>Natix</b>									
• canonical	16.0	98.6	208	79.8	470	897	166	1237	1768
• unnested	0.12	0.14	0.15	0.22	0.24	0.26	0.38	0.41	0.42

(c) Q2

Figure 7. Evaluation times (in sec.) for Q1, 2d, and Q2

commercial systems on our synthetic data set and generated two alternative query execution plans for Natix. The first alternative is based on a canonical translation (see Fig. 3(a)). The second was derived by applying a strategy based on Eqv. 4 (see Fig. 3(b)).

Figure 7(c) presents our performance measurements for these plans. The assessments indicate that all commercial systems evaluate this query similarly to our canonical plan. For the moderate size of 2.1MB of the largest synthetic data set — scaling factor 10 for both the inner and outer query block —, our unnested approach outperforms the others by three to four orders of magnitude. Moreover, evaluation times up to half an hour for 2.1MB data seem unacceptable to us.

## 5 Related Work

Apart from introducing a classification for nested queries, Kim [19] was the first to rephrase nested SQL queries to contain joins or grouping. However, the validity of these rewrites depends on important restrictions. They mainly concern empty results for the inner query block, NULL values, and duplicate handling. Subsequent research found more unnesting techniques for SQL [8, 11, 12, 18, 25], OQL [5, 9, 27, 28], and XQuery [20]. All of these techniques are restricted to conjunctive correlation or linking predicates.

Strategies for the evaluation of nested queries are discussed in [14]. However, currently the full potential for optimization is only available when queries are unnested. First results to lift this limitation are presented in [15].

[3, 16] present equivalences for rewriting quantified queries containing disjunctions. But they do not directly address query unnesting. The rewrites presented there focus on quantified queries and, hence, do not treat aggregate

functions.

To the best of our knowledge, nobody has investigated unnesting nested SQL queries with disjunctive linking or disjunctive correlation so far. Our approach for unnesting nested queries in these cases utilizes the bypass technique introduced in [17]. However, in [1], we have proposed optimization and evaluation techniques for nested XPath queries occurring in disjunctive predicates.

Because bypass operators have two output streams, which are unioned later, the resulting expression forms a directed acyclic graph (DAG). Strategies for implementing bypass operators and query evaluation engines that support DAG-structured query plans can be found in [17, 23, 24]. Especially for a thorough discussion of the generation and evaluation of DAG-structured query plans, we refer to [23].

## 6 Conclusion and Outlook

### 6.1 Conclusion

We believe that nested queries containing disjunctive predicates have not yet attracted the attention they deserve. In our experimental study, we have shown that evaluating nested queries in a nested-loop-like fashion leads to an unacceptable performance. With our novel unnesting strategy, we are able to remedy this situation and to substantially improve query execution times. Although most runtime systems and optimizers do not incorporate bypass plans, it is possible to transfer bypass plans into plans without bypass operators. This can, for example, be done by tagging every tuple whether it belongs to the positive or negative stream.

### 6.2 Outlook

Encouraged by these results, we plan to enhance our unnesting approach to handle more sophisticated cases.

These include, for example, (1) unnesting queries whose linking *and* correlation predicate occurs in a disjunction, (2) optimizing nested disjunctive queries in the *from* clause, (3) handling all linking operators, i.e.  $\theta$  ALL and  $\theta$  SOME/ANY for  $\theta \in \{<, \leq, >, \geq\}$ , and finally (4) unnesting queries featuring correlation predicates that refer to attributes defined in a non-adjacent query block (indirect correlation).

Since our unnesting technique creates DAG-structured algebraic expressions, we rely on effective optimization techniques for generating and executing DAG-structured query plans. The algebraic expressions produced by our techniques are quite demanding and, hence, might trigger further research in this direction.

**Acknowledgments** We would like to thank Simone Seeger for her comments on the manuscript, Uwe Steinel for the administration of the commercial DBMSs, and the anonymous reviewers for their helpful comments.

## References

- [1] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Kappa-join: Efficient execution of existential quantification in XML query languages. In *Proceedings of the XML Database Symposium, Seoul, Korea*, pages 1–15, 2006.
- [2] M. Brantner, N. May, and G. Moerkotte. Unnesting SQL queries in the presence of disjunction. Technical report, University of Mannheim, March 2006. <http://pi3.informatik.uni-mannheim.de/publications/TR-06-013.pdf>.
- [3] F. Bry. Towards an efficient evaluation of general queries: quantifier and disjunction processing revisited. In *Proceedings of ACM SIGMOD Conference, Oregon, USA*, pages 193–204, 1989.
- [4] B. Cao and A. Badia. A nested relational approach to processing SQL subqueries. In *Proceedings of ACM SIGMOD Conference, Baltimore, USA*, pages 191–202, 2005.
- [5] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.
- [6] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *Proceedings of 5th DBPL, Gubbio, Umbria, Italy*, 1995.
- [7] C. J. Date. The outer join. In *Proceedings of ICOD, Cambridge, England*, pages 76–106, 1983.
- [8] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the VLDB Conference, Brighton, England*, pages 197–208, 1987.
- [9] L. Fegaras. Query unnesting in object-oriented databases. In *Proceedings ACM SIGMOD Conference, Seattle, USA*, pages 49–60, 1998.
- [10] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB J.*, 11(4):292–314, 2002.
- [11] C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *Proceedings of ACM SIGMOD Conference, Santa Barbara, USA*, pages 571–581, 2001.
- [12] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *Proceedings of the ACM SIGMOD, San Francisco, USA*, pages 23–33, 1987.
- [13] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002. 0-13-098043-9.
- [14] G. Graefe. Executing nested queries. In *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Leipzig, Germany*, pages 58–77, 2003.
- [15] R. Guravannavar, H. S. Ramanujam, and S. Sudarshan. Optimizing nested queries with parameter sort orders. In *Proceedings of the VLDB Conference, Trondheim, Norway*, pages 481–492, 2005.
- [16] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.
- [17] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. *Proceedings of ACM SIGMOD Conference, Minneapolis, USA*, 23(2):336–347, June 1994.
- [18] W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.
- [19] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.
- [20] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *Proceedings of the ICDE Conference, Boston, USA*, pages 239–250, 2004.
- [21] N. May and G. Moerkotte. Main memory implementations for binary grouping. In *Proceedings of the XML Database Symposium, Trondheim, Norway*, pages 162–176, 2005.
- [22] M. Muralikrishna. Improved unnesting algorithms for join aggregate sql queries. In *Proceedings of VLDB Conference, Vancouver, Canada*, pages 91–102, 1992.
- [23] T. Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs*. PhD thesis, University of Mannheim, Germany, 2005.
- [24] P. Roy. Optimization of DAG-structured query evaluation plans. Master’s thesis, Indian Institute of Technology, Bombay, 1998.
- [25] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proceedings of the ICDE Conference, New Orleans, USA*, pages 450–458, 1996.
- [26] J. R. Slagle. An efficient algorithm for finding certain minimum-cost procedures for making binary decisions. *J. ACM*, 11(3):253–264, 1964.
- [27] H. J. Steenhagen. *Optimization of Object Query Languages*. PhD thesis, Department of Computer Science, University of Twente, 1995.
- [28] H. J. Steenhagen, P. M. G. Apers, H. M. Blanken, and R. A. de By. From nested-loop to join queries in OODB. In *Proceedings of the VLDB Conference, Santiago de Chile, Chile*, pages 618–629, 1994.
- [29] TPC. TPC benchmark H (decision support). Standard Specification Version 2.3.0, Transaction Processing Performance Council, 2006. <http://www.tpc.org/>.