The handle http://hdl.handle.net/1887/25180 holds various files of this Leiden University dissertation

**Author:** Rietveld, K.F.D.
**Title:** A versatile tuple-based optimization framework
**Issue Date:** 2014-04-10
CHAPTER 4

Forelem Extensions for Nested Queries

In this chapter we propose a way to express nested queries where subqueries are written in separate functions as forelem loop nests. Subsequently, we discuss a number of transformations which are especially suited for forelem loop nests representing nested queries. The first of these transformations is to allow a nested query to be rewritten into a single forelem loop nest. In other words, the separate functions for subqueries are inlined whenever possible. Another transformation allows for elimination of the use of temporary tables. This puts the loop nest into a form which enables further optimization using transformations as listed above. As a final transformation, we recognize canonical forms of loop nests evaluating nested queries and transformations between these forms. A transition to another canonical form has the potential to enable a whole new dimension of transformations on a loop nest.

4.1 Expressing Subqueries as Procedures

The simplest approach to express nested queries in forelem representation of the query is to write these as functions. A query containing a doubly nested query will be translated to forelem code which calls a function to perform the first subquery, which subsequently calls another function to perform the second (doubly nested) subquery.

We define a nested query to be a function containing a forelem loop nest which will execute the query at the corresponding nesting level. Any references to data or columns in the containing query must be passed as arguments to this function. Functions take zero or more arguments, which are written as $0, 1, ..., n$ in the function. The result of evaluating the nested query is returned as a return value. The SQL-92 standard [45] defines three flavors of nested queries: scalar subquery, row subquery and table subquery. This gives rise to three possibilities for return values of functions: a scalar value, a tuple or a multiset.
For example, the query

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
    FROM Reserves R
    WHERE R.bid = 103)
```

contains the nested query

```
SELECT R.sid
FROM Reserves R
WHERE R.bid = 103
```

which is written in `forelem` as:

```
function subquery()
{
    T = ∅
    forelem (i; i ∈ pR.bid[103])
        T = T ∪ (R[i].sid)
    return T
}
```

where `T` is a temporary table. It is clear that this concerns a table subquery. For scalar subqueries and row subqueries, the return value is simply replaced with a scalar or a tuple respectively.

To see how functions with arguments are handled, let us consider the following query:

```
SELECT R.date
FROM Reserves R
WHERE R.bid = 103 AND EXISTS (SELECT S.sname
    FROM Sailors S
    WHERE S.sid = R.sid
    AND S.sname = "Horatio")
```

where the value `R.sid` is passed to the subquery. The procedure performing the nested query is written as:

```
function subquery($0)
{
    T = ∅
    forelem (i; i ∈ pS.(sid,sname)[$0,"Horatio"])
        T = T ∪ (S[i].sname)
    return T
}
```

The main query will iterate the `Reserves` table with an induction variable `i` and pass the value of `R[i].sid` as the first argument to the procedure `subquery`. 
4.2 Expressing Nested Query Operators in \textit{forelem}

At least as important is the problem of how to express the different operators that can be used in conjunction with the results of nested queries expressed in \textit{forelem} loops. Such operators include \texttt{IN}, \texttt{EXISTS}, \texttt{ANY}, etc. Let us consider the \texttt{IN} operator used in our example:

\begin{verbatim}
S.sid IN (SELECT R.sid
FROM Reserves R
WHERE R.bid = 103)
\end{verbatim}

This predicate evaluates to true if \texttt{S.sid} is in the set resulting from the evaluation of the nested query. Bearing in mind that we can generate an index set for each table in a \textit{forelem} loop nest, including temporary tables, we can write the entire query using \textit{forelem} constructs as follows:

\begin{verbatim}
function subquery()
{
  \mathcal{T} = \emptyset
  forelem (i; i \in pR.bid[103])
    \mathcal{T} = \mathcal{T} \cup (R[i].sid)
  return \mathcal{T}
}

forelem (i; i \in pS)
{
  \mathcal{T} = subquery()
  forelem (j; j \in is_not_empty(p.T.sid[S[i].sid]))
    R = R \cup (S[i].sname)
}
\end{verbatim}

In this code fragment, \texttt{is_not_empty} is an index set modifier. The purpose of an index set modifier is to modify the index set without touching the original, so that the iteration space of the \textit{forelem} loop is changed. For example, the \texttt{single} modifier modifies the iteration space such that only the first index from the index set is returned. The main advantage of modifiers is that they can be embedded in a \textit{forelem} statement, such that the exact iteration sequence is still encapsulated and \textit{forelem} statements can participate in the various \textit{forelem} loop nest transformations which have been defined. Also, modifiers can be reduced by introducing a mask column in the table. We have discussed mask columns earlier in Section 3.3.5.

The \texttt{is_not_empty} modifier will return the first index of the index set when the index set is non-empty, otherwise nothing is returned which will refrain the \textit{forelem} loop from executing its loop body. In fact, the modifier is equal in operation to the \texttt{single} modifier, which also returns the first index, but for clarity we maintain both.

The \texttt{is_empty} modifier, with the inverse operation, also exists. In this case, however, a dummy index is returned instead of the first index into the index set (since the index set is empty). The dummy index cannot be used to access a table.
For this particular code fragment, the is_not_empty modifier will cause the loop body of the forelem statement to either be executed once or not be executed at all. Only a single tuple is added to the result set or none.

NOT IN can be implemented by simply replacing the is_not_empty modifier with is_empty. The keyword EXISTS tests whether a relation is not empty and can be implemented similar to the example. In this case, the index set $p.T$ will not have a condition attached.

Another class of nested query operators are the ANY and ALL operators. These operators are combined with an inequality operator such as $<$ or $>$. A construction such as

\[
S.\text{rating} > \text{ALL} \left( \right. \text{SELECT} \ S2.\text{sid} \\
\left. \text{FROM} \ Sailors \ S2 \right.
\text{WHERE} \ S2.\text{snmae} = "\text{Horatio}"
\]

evaluates to true if $S.\text{rating}$ is larger than all ratings of sailors named "Horatio". Or, put differently, $S.\text{rating}$ is larger than the largest rating of a sailor named "Horatio". From this follows that it is possible to express nested queries using ANY and ALL as nested queries which compute a MAX or MIN aggregate and vice versa, as is described in [36].

Let us consider the query using $> \text{ALL}$ in full:

\[
\text{SELECT} \ S.\text{sid} \ \text{FROM} \ Sailors \ S \ \\
\text{WHERE} \ S.\text{rating} > \text{ALL} \left( \right. \text{SELECT} \ S2.\text{rating} \\
\left. \text{FROM} \ Sailors \ S2 \right.
\text{WHERE} \ S2.\text{snmae} = "\text{Horatio}"
\]

The query is to return the sid of all sailors who have a rating larger than all sailors named "Horatio". We will employ the is_empty modifier again to express this query using forelem loops:

\[
\text{function} \ \text{subquery}() \\
\{
  T = \emptyset \\
  \text{forelem} \ (i; \ i \in p.\text{name}["\text{Horatio}"])
  \ T = T \cup (S[i].\text{rating})
  \text{return} \ T
\}
\]

\[
\text{forelem} \ (i; \ i \in p.\text{S}) \\
\{
  T = \text{subquery}() \\
  \text{forelem} \ (j; \ j \in \text{is_empty}(p.T.\text{rating}([S[i].\text{rating}, \infty])))
  \ R = R \cup (S[i].\text{sid})
\}
\]

Please note that the notation is intermixed with $[ , )$ denoting closed/open intervals. We have seen the notation rating[18] earlier, which denotes that the value of the column rating must equal 18. The interval notation specifies that the value
of rating must be in the specified interval. In this case, the interval \([S[i].rating, \infty)\) denotes that rating should be equal or larger than \(S[i].rating\).

The condition in the query that \(S[i].rating\) must be greater than the ratings of all sailors named "Horatio" (or rather greater than the highest rating of a sailor named "Horatio") is translated to: there exists no rating for a sailor named "Horatio" which is larger or equal to \(S[i].rating\). We test whether the index set on the temporary relation returned by the subquery is empty if we select on ratings equal to or larger than \(S[i].rating\). If this index set is empty, we will output a single tuple by performing a single iteration of the inner loop body.

Similarly, \(< \text{ ALL} \) translates to checking whether a value is smaller than all values returned by a nested query (or smaller than the minimum value, e.g. the use of a \(\text{MIN}\) aggregate).

Where \(> \text{ ALL}\) compares whether a given value is larger than all values (or the maximum) of a given set, \(> \text{ ANY}\) compares whether a given value is larger than any value (or the minimum) of a given set. Essentially, maximum is swapped for minimum. Therefore, it is possible to express \(\text{ANY}\) queries using \(\text{forelem}\). Let's consider the following example:

```sql
SELECT S.sid FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
    FROM Sailors S2
    WHERE S2.sname = "Horatio")
```

which can be written using \(\text{forelem}\) loops as follows:

```plaintext
function subquery()
{
    \(T = \emptyset\)
    \(\text{forelem } (i; i \in \text{pS.name["Horatio"]})\)
    \(T = T \cup (S[i].rating)\)
    \(\text{return } T\)
}

\(\text{forelem } (i; i \in \text{pS})\)
{
    \(T = \text{subquery}()\)
    \(\text{forelem } (j; j \in \text{is_not_empty(pT.rating((-\infty, S[i].rating)])))}\)
    \(R = R \cup (S[i].sid)\)
}
```

The selection logic was transformed to say that there must be a rating in \(T\) which is smaller than or equal to \(S[i].rating\), otherwise \(S[i].rating\) can never be larger than any rating in \(T\).

Although nested queries with \(\text{ANY}\) and \(\text{ALL}\) can be expressed as \(\text{forelem}\) loops without problems, in the literature it is suggested to use a nested query using a \(\text{MIN}\) or \(\text{MAX}\) aggregate function instead to avoid potential confusion [82]. For the last query discussed, \(S[i].rating\) must be larger than the minimum rating found for a sailor named "Horatio". This can be written in SQL as follows:
SELECT S.sid FROM Sailors S
WHERE S.rating > (SELECT MIN(S2.rating)
FROM Sailors S2
WHERE S2.sname = "Horatio")

Note that this subquery returns a scalar value and thus is a scalar subquery, contrary to the ANY and ALL queries which use table subqueries. This subquery is similarly expressed as a function which returns a scalar value. For the implementation of aggregate functions using forelem we refer to Chapter 5.

4.3 Set Operators

The SQL standard [45] also describes set operators, such as UNION, INTERSECT and EXCEPT. Queries using these set operators could be considered nested queries as well, because the query actually consists out of more than one subquery. While SQL queries retain duplicates by default (unless the DISTINCT keyword is used), it is important to understand that the set operators do not retain duplicates by default.

Let us consider two temporary tables \( T_1 \) and \( T_2 \). To implement the UNION set operator, we merge the two temporary tables and make sure each tuple only appears once. The query can be expressed in forelem loops using the distinct syntax\(^1\), that will be introduced in Section 5.2, and the is_empty modifier introduced in this chapter:

\[
T_1 = \text{subquery1}() \\
T_2 = \text{subquery2}() \\
\text{forelem} (i; i \in pT_1.\text{distinct}) \\
T = T \cup (T_1[i]) \\
\text{forelem} (i; i \in pT_2.\text{distinct}) \\
\text{forelem} (j; j \in \text{is_empty}(pT_1[T_2[i]])) \\
T = T \cup (T_2[i])
\]

We start by adding all distinct tuples from \( T_1 \) to the result set. After that, we add all distinct tuples from \( T_2 \) which do not appear in \( T_1 \). It is clear that a query in a subquery can be inlined in this loop nest. The temporary tables can possibly be eliminated using the transformation described in Section 4.4.2

The EXCEPT operator can be implemented using the same constructs. The goal is to include all tuples in the result set which appear in \( T_1 \) but not in \( T_2 \). This results in the following code:

\[
T_1 = \text{subquery1}() \\
T_2 = \text{subquery2}() \\
\text{forelem} (i; i \in pT_1.\text{distinct})
\]

\(^1\)An index set can be suffixed with the keyword .distinct. This has as result that the index set only consists of unique tuples, any duplicate tuple is excluded from the iteration space.
4.4 Transformations on Nested Queries

We described several transformations for *forelem* loop nests in Section 3.3. These transformations include Loop Collapse, Loop Interchange and Table Reduction Operators. Using these transformations, the loop nest can be transformed into a more optimal code or a better table layout can be discovered by applying Loop Collapse and Reverse Loop Collapse.

The more loops a *forelem* loop nest contains, the more effective these transformations are. Therefore, for the nested queries discussed in this chapter it is of interest to inline the nested queries such that a single large loop nest exists. In this section we will present transformations to accomplish this and transformations to introduce and eliminate temporary storage. Whether or not temporary storage should be used is a trade-off between memory usage, cache usage and the time needed for calculating the contents of the temporary storage.

### 4.4.1 Transforming a Nested Query to a Single *forelem* Loop Nest

Let us consider a singly-nested query, in which we can identify two subqueries $S_1$ and $S_2$. This is done according to the form $S_1 \text{ WHERE } \text{field IN (} S_2 \text{)}$, but also works for operators other than IN. As an example, we will re-use the query:

```sql
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                 FROM Reserves R
                 WHERE R.bid = 103)
```

which was introduced earlier in this chapter, where we also presented this query expressed in *forelem* loops. When we inline the procedure performing the sub-query in *forelem* loops, we obtain:

```c
forelem (j; j ∈ is_empty(\text{p}\mathcal{T}_2[\mathcal{T}_1[i]]))
\mathcal{R} = \mathcal{R} \cup (\mathcal{T}_1[i])
```

Note that this is similar to a query using NOT IN. The INTERSECT operator is the opposite and can be compared to a query using the IN operator:

```c
\mathcal{T}_1 = \text{subquery1}()
\mathcal{T}_2 = \text{subquery2}()
```

```c
forelem (i; i ∈ p\mathcal{T}_1.distinct)
forelem (j; j ∈ is_not_empty(\text{p}\mathcal{T}_2[\mathcal{T}_1[i]]))
\mathcal{R} = \mathcal{R} \cup (\mathcal{T}_1[i])
```
forelem (k; k ∈ pR.bid[103])
    T = T ∪ (R[k].sid)
forelem (j; j ∈ is_not_empty(pT.sid[S[i].sid]))
    R = R ∪ (S[i].sname)
}

It is now clearly visible that the index set pR.bid[103] is generated for each iteration in the inner loop. The SQL standard [45] mandates that the subquery is effectively executed for each evaluation of the WHERE clause of the main query. In this case it is obvious that this particular index set, and the resulting table T after iteration of this index set, will be the same in each iteration of the outer loop (of course as long as the table Reserves is not modified). We therefore argue that if the loop generating T is only performed once, outside of the outer loop, it is still effectively executed for each evaluation of the WHERE clause because the intended result, which is always the same for every loop iteration, is always used for each clause evaluation. So, when an analysis can prove that index sets are loop invariant (the subquery is not corresponding) and are free from side effects, more aggressive optimization is possible by moving loop invariant code segments.

4.4.2 Temporary Table Reduction

It will often be the case that a forelem loop is producing tuples into a temporary table, which is only read by a consecutive forelem loop. In such cases it might be beneficial to eliminate the usage of the temporary table and to merge the two forelem loops. This transformation is different from the Loop Merge transformation described in Section 3.3.4, as that particular transformation targets a forelem loop and while loop. We distinguish two cases of Temporary Table Reduction.

Let us first consider the simple case, in which we have two forelem loops, one producing tuples into a temporary table and the second loop fully iterating over this temporary table:

T = ∅
forelem (k; k ∈ pR.bid[103])
    T = T ∪ (R[k].sid)
forelem (j; j ∈ pT)
    forelem (i; i ∈ pS.sid[T[j].sid])
        R = R ∪ (S[i].sname)

In this case it is obvious that we can merge the forelem loop with iteration counter j with the forelem loop with iteration counter k that produces the tuples that are consumed. Note that the intention here is to merge two forelem loops, contrary to the Loop Merge transformation for forelem loops described in Section 3.3, where the loop body of a while loop iterating the results is merged into the body of the inner loop of a forelem loop nest. The resulting code is:

forelem (k; k ∈ pR.bid[103])
    forelem (i; i ∈ pS.sid[R[k].sid])
        R = R ∪ (S[i].sname)
and the temporary table has been eliminated. The reverse operation, Temporary Table Introduction follows from this by introducing a temporary table and having a \textit{forelem} loop add tuples to this table containing all fields which are used by the \textit{forelem} loops that will consume the tuples from this temporary table.

We now consider a different example where conditions are present on the index set on the temporary table:

$$\mathcal{T} = \emptyset$$

\textbf{forelem} (k; k \in pR)
$$\mathcal{T} = \mathcal{T} \cup (R[k].sid, R[k].bid)$$

\textbf{forelem} (j; j \in p\mathcal{T}.bid[103])
$$\text{if } (\mathcal{T}[j].bid == 103)$$
$$\textbf{forelem} (i; i \in pS.sid[\mathcal{T}[j].sid])$$

\textbf{R} = \textbf{R} \cup (S[i].sname)

Now the temporary table can be eliminated analogously to the first example. The reference \(\mathcal{T}[j].bid\) in the \textit{if}-statement will be written as \(R[k].bid\). Finally, the \textit{if}-statement can simply be transformed into a condition on the index set on Reserves, making the resulting code fragment equal to the result of the transformations on the first example.

### 4.4.3 Modifier Manipulation

We cannot apply Temporary Table Reduction to the example we introduced in Section 4.4.1, because the loop we would like to merge contains an \textit{is_not_empty} modifier. When an index set is wrapped in a modifier, it is not immediately possible to simply move the conditions to \textit{if}-statements. To correctly evaluate a modifier such as \textit{is_not_empty}, it is necessary to know the final size of the index set after the conditions have been applied. In order to enable application of the Temporary Table Reduction transformation in such cases, we introduce two transformations to manipulate index sets with conditions within a modifier.

In the example, we observe that the full iteration of index set \(pR.bid[103]\) generates the table \(\mathcal{T}\). This table is only used by the index set \(p\mathcal{T}.sid[S[i].sid]\). In fact, the first index set is being further narrowed down by selecting a specific \(sid\). Instead of narrowing down on this specific \(sid\) within the modifier, we can transfer this condition into a preceding \textit{forelem} loop as follows:

\textbf{forelem} (i; i \in pS)

\[ T = \emptyset \]
\[
\text{forelem } (k; k \in pR.bid[103])
\]
\[
T = T \cup (R[k].sid)
\]
\[ T_2 = \emptyset \]
\[
\text{forelem } (q; q \in pT.sid[S[i].sid])
\]
\[
T_2 = T_2 \cup T[q]
\]
\[
\text{forelem } (j; j \in \text{is_not_empty}(pT_2))
\]
\[
R = R \cup (S[i].sname)
\]

The condition on the index set iterated by the variable \( q \) can now be moved to an \textit{if}-statement within the loop body. After that, the temporary table \( T \) can be eliminated, resulting in:

\[
\text{forelem } (i; i \in pS)
\]
\[
\{
\text{forelem } (k; k \in pR.bid[103])
\]
\[
\text{if } (R[k].sid == S[i].sid)
\]
\[
T_2 = T_2 \cup R[k]
\]
\[
\text{forelem } (j; j \in \text{is_not_empty}(pT_2))
\]
\[
R = R \cup (S[i].sname)
\]

and after moving the \textit{if} condition into the index set:

\[
\text{forelem } (i; i \in pS)
\]
\[
\{
\text{forelem } (k; k \in pR.(bid,sid)[(103,S[i].sid)])
\]
\[
T_2 = T_2 \cup R[k]
\]
\[
\text{forelem } (j; j \in \text{is_not_empty}(pT_2))
\]
\[
R = R \cup (S[i].sname)
\]

When we observe that the index set \( pT_2 \) is never iterated and only passed to the \textit{is_not_empty} modifier, we can merge the two remaining inner loops, resulting in the compact form:

\[
\text{forelem } (i; i \in pS)
\]
\[
\text{forelem } (j; j \in \text{is_not_empty}(pR.(bid,sid)[(103,S[i].sid)]))
\]
\[
R = R \cup (S[i].sname)
\]

The code fragment will output a single tuple for each \( S[i].sid \) which has reserved a boat with \textit{bid} 103. This equals the original SQL expression. As a result, we have successfully transformed the original nested query to a single perfectly nested forelem loop nest. Note that this loop nest looks similar to how joins are expressed in forelem, we will discuss this further in Section 4.4.5.
4.4.4 Canonical Forms for Nested Queries

In the previous sections we discussed how a nested query, with its subqueries expressed as functions, can be transformed to a single loop nest and how further transformations are possible on a single loop nest. We highlighted in Section 4.4.1 that transformations that move the execution of the “subquery loop” are possible when we can prove through dependency analysis that the moved code is invariant to the enclosing loop.

When we consider singly-nested queries of which the subquery is loop invariant such that it can be subjected to various transformations, we can define four canonical forms for a singly-nested query with a main query $S_1$ and subquery $S_2$. These four forms are shown in Figure 4.1. The starting point, where the subquery is inlined in the body of the loop of the main query, is form 1.

$$\begin{align*}
1) \text{forelem } S_1 & \quad 2) \text{forelem } S_1 \quad 3) \text{forelem } S_2 & \quad 4) \text{forelem } S_2 \\
\text{forelem } S_2 & \quad | & \quad \text{forelem } S_1 & \quad | & \quad \text{forelem } S_1 \\
\text{forelem } S_2 & & \text{forelem } S_1
\end{align*}$$

Figure 4.1: Four canonical forms for nested queries expressed as \textit{forelem} loop nest expressed as \textit{forelem} loop nests.

Loop nest forms 1) and 3) can be transformed into one another using the Loop Interchange transformation for \textit{forelem} loop nests defined Section 3.3.2. The forms 2) and 4) store the results of $S_1$ or $S_2$, respectively, in a temporary table. Form 1) can simply be transformed into form 2) by first running the outer loop and storing the results in a temporary table. After that, the inner loop is ran. Basically, this is a transformation which introduces a temporary table in the computation. The reverse, the elimination of a temporary table is also possible. We described this transformation in Section 4.4.2.

By transforming between different canonical forms the search space for the most efficient loop nest is increased. Different canonical forms enable a larger number of possible transformations that can be applied. For example, by moving the subquery loop out of the main loop, such as in form 4), the opportunity is created to possibly merge the subquery loop with any loop preceding it. This can only be done if they operate on the same table. For certain loop nests, however, only moving the subquery loop out of the main loop and storing the results in temporary storage might already be beneficial. Another advantage of the canonical forms is that they allow for loop interchanges; different iteration orders have a different impact on cache utilization.

We have considered the canonical forms for a singly-nested subquery in this section. Using these forms as base case, canonical forms for doubly-nested subqueries and further nesting levels can be considered. Also, forms can be considered for sequences of singly-nested subqueries or subqueries with mixed depth. Naturally, this will result in a larger number of forms, which again implies a larger search space for potentially more efficient loop nests.

Kim [53] identified five basic types of nested queries and described transformations to go from a nested query to a canonical form, which is a canonical n-relation query. The described transformations allow nested queries of arbitrary
depth to be transformed. Ganski et al. present solutions for bugs identified in Kim’s transformations, as well as extend the transformations with the ability to handle more nested query operators [36]. The canonical form used by Kim is different from the canonical forms we described in this section. Kim’s canonical form is used as a target to transform to, because query processing systems (at that time) can generally process single-level queries performing joins more efficiently than nested queries. Our canonical forms for subqueries are used to extend the search space for efficient loop nests. By transforming a loop nest into a different canonical form, another set of transformations can be applied eventually leading to more efficient code.

4.4.5 Relationship Between IN Nested Query Operators and Joins

The possibility to express a nested query using the IN operator as a query using a join is well known [53, 36]. However, the original approach described in [53] relies on the assumption that duplicates are to be removed from the results of executing the subquery, in order for the nested query and the query using a join to produce the same final results.

With the queries expressed in forelem loop nests, this subtle difference becomes very clear. Recall that at the end of Section 4.4.2 we observed that we transformed a nested query using the IN operator into a perfectly nested forelem loop, similar to how joins are expressed in forelem loop nests. A notable difference is the use of is_not_empty in the inner loop. This difference is clearly visible when we compare the loop of Section 4.4.2 with the perfectly nested loop nest representing a join query:

1) \[
\text{forelem } (i; \ i \in pS) \\
\text{forelem } (j; \ j \in \text{is_not_empty}(pR.(bid,sid)[(103,S[i].sid)])) \\
R = R \cup (S[i].sname)
\]

2) \[
\text{forelem } (i; \ i \in pS) \\
\text{forelem } (j; \ j \in pR.(bid,sid)[(103,S[i].sid)]) \\
R = R \cup (S[i].sname)
\]

These loops correspond to the following two queries:

1) \[
\text{SELECT } S.sname \\
\text{FROM } Sailors S \\
\text{WHERE } S.sid \text{ IN } (\text{SELECT } R.sid \\
\text{FROM } Reserves R \\
\text{WHERE } R.bid = 103)
\]

2) \[
\text{SELECT } S.sname \\
\text{FROM } Sailors S, Reserves R \\
\text{WHERE } S.sid = R.sid \\
\text{AND } R.bid = 103
\]

Although both queries answer the same “question”, listing all sailor names who have reserved the boat with bid 103, the generated result tables are slightly different. Query 1) will output a sailor’s name once if its sid appears in the list of sids which have reserved boat 103. Query 2) will output a sailor’s name for each reservation of boat 103 by that sailor, a sailor’s name appears as many times as the sailor reserved boat 103. In other words, in query 1) duplicate reservations
are omitted; query 1) yields a subset of query 2). The duplicate reservations are omitted due to the use of a nested query and the IN operator. Note that if two sids exist with the same sailor name who both have reserved boat 103, then this sailor name will still be duplicated!

This difference can be seen in loop nest 1) too. Due to the use of is_not_empty, the inner loop is only ran once if a sailor has reserved boat 103; so the sailor name is only output once regardless of the number of reservations. Loop nest 2) can be modified to produce the same result as loop nest 1) easily. The generation of duplicate sids has to be stopped. To constrain this loop to a single iteration, we use the single or the equal is_not_empty modifier. When these modifiers are used, the loop becomes equal to loop nest 1).

4.5 Example

The following code fragment is based on code taken from Discus. Discus is a web application which has been developed in-house at LIACS and is a complete solution for administration of students, courses, exams and programs. The code fragment is written in pseudocode similar to PHP and edited for clarity. The code is based on a function which generates a listing of courses for which a given student (whose ID is specified through parameter $id) is registered. For each course, the main teacher as well as any additional teachers are retrieved.

```php
$coursesResult = mysql_query('SELECT * FROM courses, teachers ' . 
  'WHERE courses.id IN (select course_id FROM students_courses ' . 
  'WHERE student_id = ' . $id . ') AND teacher_id = teachers.id');

while ($row = mysql_fetch_row($coursesResult))
{
  $out = array();
  for ($i = 0; $i < mysql_num_fields($coursesResult); $i++)
  {
    $column = mysql_fetch_field($coursesResult, $i);
    $out[$column->table][$column->name] = $row[$i];
  }
  $courses[] = $out;
}

$i = 0;
foreach ($courses as $course)
{
  $teachersResult = 
    mysql_query('SELECT * FROM teachers, teachers_courses ' . 
    'WHERE teachers.id = teachers_courses.teacher_id AND ' . 
    'teachers_courses.course_id = ' . 
    $courses[$i]['courses']['id']);
  while ($row = mysql_fetch_row($teachersResult))
  {
    $out = array();
```
**Forelem Extensions for Nested Queries**

```php
for ($i = 0; $i < mysql_num_fields($teachersResult); $i++)
{
    $column = mysql_fetch_field($teachersResult, $i);
    $out[$column->table][$column->name] = $row[$i];
}
$teachers[] = $out;

$j = 0;
foreach ($teachers as $teacher)
{
    $courses[$i]['teachers'][$j] = $teacher['teachers'];
    $j++;
}
$i++;
}
```

As a first step, we replace the usage of the DBMS API with *forelem* loop nests and result sets. We obtain:

```php
function subquery($0)
{
    \mathcal{T} = \emptyset
    forelem (i; i \in pStudents_courses.student_id[$0])
        \mathcal{T} = \mathcal{T} \cup (students_courses[i].course_id)
    return \mathcal{T}
}

forelem (i; i \in pCourses)
    forelem (j; j \in pTeachers.id[courses[i].teacher_id])
        \mathcal{T} = subquery($id)
        forelem (k; k \in is_not_empty(p\mathcal{T}.course_id[courses[i].id]))
            \mathcal{R}_1 = \mathcal{R}_1 \cup (courses[i].*, teachers[j].*)
    while ($row \in \mathcal{R}_1)
    {
        $out = array();
        for ($i = 0; $i < len($row); $i++)
        {
            ($table, $column) = get_field_info($row, $i)
            $out[$table][$column] = $row[$i];
        }
        $courses[] = $out;
    }
```
$i = 0;

foreach ($courses as $course)
{
    $course_id = $courses[$i]['courses']['id'];

    forelem (i; i ∈ pTeachers_courses.course_id[$course_id])
    
    forelem (j; j ∈ pTeachers.id[teachers_courses[i].teacher_id])

    $R_2 = R_2 \cup (teachers_courses[i].*, teachers[j].*)

while ($row ∈ R_2)
{
    $out = array();

    for ($i = 0; $i < len($row); $i++)
    {
        ($table, $column) = get_field_info($row, $i)
        $out[$table][$name] = $row[$i];
    }
    $teachers[] = $out;
}

$j = 0;

foreach ($teachers as $teacher)
{
    $courses[$i]['teachers'][$j] = $teacher['teachers'];
    $j++;
}

$i++;
}

In the code fragment a function get_field_info is used to get field information. The $row variable is a reference to a tuple generated by a forelem loop; this tuple does contain information on the tables and columns from which the tuple members originated.

Focusing on the first forelem loop nest, we note that the parameter passed to the subquery does not depend on the state of the evaluation of the main query. Due to the absence of such dependencies, the query is non-corresponding and can be evaluated separately. See also the discussion in Section 4.4.1. Therefore, it is possible to inline the function subquery into the main loop nest:

forelem (i; i ∈ pCourses)

forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
{
    $T = ∅

    forelem (l; l ∈ pStudents_courses.student_id[$id])
    $T = T \cup (students_courses[l].course_id)

    forelem (k; k ∈ is_not_empty(pT.course_id[courses[i].id]))
    $R_1 = R_1 \cup (courses[i].*, teachers[j].*)
}
The `forelem` loop with iteration counter 1 generates a temporary table which is immediately consumed by the `forelem` loop with iteration counter k. This `forelem` loop uses the index set on the temporary table within a modifier. To be able to apply Temporary Table Reduction as described in Section 4.4.2, we first move the conditions on this index set out of the modifier using the transformation described in Section 4.4.3. The first step is to push the condition in the `forelem` loop over $pT$ to an `if`-statement:

```java
forelem (i; i ∈ pCourses)
  forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
  {
    $T = \emptyset$
    forelem (l; l ∈ pStudents_courses.student_id[$id])
      $T = T \cup (students_courses[l].course_id)$
    $T_2 = T \emptyset$
    forelem (q; q ∈ pT.course_id[courses[i].id])
      $T_2 = T_2 \cup T[q]$
    forelem (k; k ∈ is_not_empty(pT2)
      $R_1 = R_1 \cup (courses[i].*, teachers[j].*)$
  }
```

Now that the inner loops are in the form discussed in Section 4.4.2, we apply similar steps to eliminate usage of both $T_1$ and $T_2$.

```java
forelem (i; i ∈ pCourses)
  forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
    forelem (k; k ∈ is_not_empty(pStudents_courses. (student_id,course_id)[($id, courses[i].id)])
      $R_1 = R_1 \cup (courses[i].*, teachers[j].*)$
```

If we subsequently merge the `while` loop consuming tuples from $R_1$ with the obtained `forelem` loop nest, the result is:

```java
forelem (i; i ∈ pCourses)
  forelem (j; j ∈ pTeachers.id[courses[i].teacher_id])
    forelem (k; k ∈ is_not_empty(pStudents_courses. (student_id,course_id)[($id, courses[i].id)])
      {
        $row = (courses[i].*, teachers[j].*)$
        $out = array();$
        for ($i = 0; $i < len($row); $i++)
          {
            ($table, $column) = get_field_info($row, $i)
            $out[$table][$column] = $row[$i];
          }
        $courses[] = $out;
      }
```

We now turn our attention to the second query:

\[
\text{forelem} \ (i; \ i \in \ p\text{Teachers_courses.course_id}[\$course_id]) \\
\text{forelem} \ (j; \ j \in \ p\text{Teachers.id}[\text{teachers_courses[i].teacher_id}]) \\
R_2 = R_2 \cup (\text{teachers_courses[i].*}, \text{teachers[j].*})
\]

On this query we will apply the Loop Collapse transformation as described in Section 3.3.5. Important is that the generated cross product is intermediate and will not be generated in full during the code generation phase if it is determined that generating the cross product will be more expensive than an evaluation of the query using two separate tables. More transformations might be done, for example to remove unused columns and rows from the cross product using the Horizontal Iteration Space Reduction and Vertical Iteration Space Reduction transformations described in Sections 3.3.7 and 3.3.8 respectively. Furthermore, the use of `get_field_info` on the collapsed table will continue to refer to the original tables, not the collapsed table, for compatibility.

Next to the Loop Collapse transformation, we also merge the `while` loop consuming the results from \(R_2\) into the loop body where these results are generated.

\[
\text{forelem} \ (i; \ i \in \ p\text{Teachers_courses} \times \text{Teachers.(course_id, id, Teachers.Courses)}, (\text{course_id, teacher_id, Teachers.Courses})) \\
\text{forelem} \ (j; \ j \in \ p\text{Teachers.id}[\text{teachers_courses[i].teacher_id}]) \\
\text{forelem} \ (k; \ k \in \ \text{is_not_empty}(\ p\text{Students_courses}. \ (\text{student_id, course_id}))[\text{($id, courses[i].id$})])
\]

\[
\text{forelem} \ (i; \ i \in \ p\text{Courses}) \\
\text{forelem} \ (j; \ j \in \ p\text{Teachers.id}[\text{courses[i].teacher_id}]) \\
\text{forelem} \ (k; \ k \in \ \text{is_not_empty}(\ p\text{Students_courses}. \ (\text{student_id, course_id}))[\text{($id, courses[i].id$})])
\]

Finally, we will look at the example as a whole again. We will merge the two loops following each `forelem` loop, that is one loop iterating over the \$courses array generated by the first `forelem` loop and another loop iterating over the \$teachers array generated by the second `forelem`. We obtain a single loop nest:

\[
$i = 0; \\
\text{forelem} \ (i; \ i \in \ p\text{Courses}) \\
\text{forelem} \ (j; \ j \in \ p\text{Teachers.id}[\text{courses[i].teacher_id}]) \\
\text{forelem} \ (k; \ k \in \ \text{is_not_empty}(\ p\text{Students_courses}. \ (\text{student_id, course_id}))[\text{($id, courses[i].id$})])
\]

\[
$i = 0; \\
\text{for} \ (i = 0; \ i < \text{len}($row); \ i++)
\]

($table, $column) = get_field_info($row, $i)
$out[$table][$column] = $row[$i];
}
$courses[] = $out;

$j = 0;
$course_id = $courses[$i]['courses']['id'];
forelem (ii; ii ∈ pTeachers_courses × Teachers.
    (course_idTeacher_Courses, idTeachers)
    [($course_id, teacher_idTeacher_Courses)])
{
    $row = Teachers_courses × Teachers[ii];
    $out = array();
    for ($ii = 0; $ii < len($row); $ii++)
    {
        ($table, $column) = get_field_info($row, $ii)
        $out[$table][$name] = $row[$ii];
    }
    $teachers[] = $out;

    $courses[$i]['teachers'][$j] = $teacher['teachers'];
    $j++;
}
$i++;
}

Further transformations are certainly possible on this loop nest. The calls to get_field_info in the final code fragment are really the remains of the use of a SQL API in the original code. Because we have knowledge about the table and column names within the code, we can simply unroll the loops performing these calls. This will result in direct assignments of table data to the output rows, for example:

$out['courses']['id'] = $row[0];

After this loop unroll, it becomes easier to trace back the assignment statement corresponding to uses of the $courses array using def-use analysis. Def-use analysis can also detect that the subscripts of the $courses array are appended by adding $out variables. The use of the $out variable can be eliminated by immediately appending it to the $courses array, a transformation which further simplifies the code.

This also paves the way for elimination of assignments from the table data to the $courses array for array items, which are never accessed. For example, for the inner forelem loops over Teachers and Teachers_Courses, we will be able to detect that the Teachers_Courses fields are never accessed. This information can be used to eliminate the unused Teachers_Courses fields from the cross product.

The simplified code after def-use analysis also has the potential to involve the inner two forelem loops with the outer three forelem loops. Further application of
the Loop Collapse or Loop Interchange transformations could be possible, eventually leading to further data reformatting.

## 4.6 Conclusions

We have expanded the syntax and set of transformations of `forelem` loops introduced in the previous chapter, with syntax and transformations to handle nested queries. These transformations simplify the analysis of nested queries. So established techniques such as dependency analysis can be used to determine whether or not a subquery has to be executed for every evaluation of the `WHERE` clause, or whether a single execution of the subquery is sufficient to be able to comply with the SQL-92.

By means of an example, we have demonstrated that many potential optimizations exist, that can take advantage of the described transformations. In the example, we were able to merge two separate queries and two separate loops processing the query results into one small and concise loop that is semantically equivalent. Still, there are possibilities to further optimize this loop nest.