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
The previous chapter described an initial study on overhead in web applications showing that 90% of the instructions executed to generate web pages are non-essential; in other words, these can be eliminated without affecting the final result. This could result in a saving of energy consumption by computer hardware in data centers between 70% and 95%. A large part of the 90% reduction of the amount of executed instructions is due to integration of the code to evaluate database queries into the application code. By the combined integration of application codes and the database requests, it becomes possible to optimize applications by optimizing compiler technology after the division between application and DBMS codes has been eliminated.

To help this integration, we propose a methodology to efficiently express database queries in terms of an imperative language and thus allowing for integration of the application code with the code performing the evaluation of the database query. Within this methodology, queries are expressed in such a way that full integration in the work flow of common optimizing compilers is achieved. This makes it possible to unleash the full power of optimizing compilers on the combination of application and database codes.

Current development environments and frameworks to develop, for example, Java-based database applications or PHP-based web applications, have been serving programmers very well. They enabled programmers to rapidly develop and deploy complex web applications. Without these technologies, the World-Wide Web would not have made such a large advancement as it did in the last decade. So, whatever change we are proposing to improve DBMS performance, this development methodology should be kept in place.

To eliminate the observed overhead, our aim is to develop a code optimization backend, or global integrated optimization process, that is able to take an existing database or web application and automatically breaks down the layers that incur overhead. This code optimization backend will co-exist with contemporary development methods and frameworks for web applications. An application is
developed and tested as usual, but before extensive deployment in a data center, the code is passed through the code optimization backend to eliminate as much overhead as possible. This way, we continue to take advantage of the available software development tools which enhance programmer productivity and combine this with a code optimization backend that significantly improves the performance of the application and reduces the energy consumed by the hardware which runs the application.

In order to realize the code optimization backend, a methodology is needed to efficiently express database queries in terms of an imperative language and that allows for integration of application code with the code performing the evaluation of the database query. Within this methodology, the database queries must be expressed in such a way which allows for full integration in the work flow of common optimizing compilers. In this chapter, we introduce such a method: forelem loop nests. forelem loop nests are designed to integrate DBMS queries in a normal optimizing compiler work flow, and also support many database-style optimizations such as the use of various kinds of join algorithms. It is important to note that forelem loop nests will only be used by the code optimization backend and it is explicitly not our intention to present forelem as a new programming methodology to write database applications.

This chapter demonstrates that simple SQL queries can be expressed in terms of forelem loops. In Chapters 4 and 5 extensions are proposed so that nested queries, aggregate functions and group-by queries can be expressed. When a query is expressed in the forelem framework, the complexities of query evaluation are encapsulated and what remains is a collection of simple loops. As will be described in this chapter, such loop nests are very well suited for optimization by optimizing compilers and a number of transformations will be described that can be applied on forelem loops nests, such as loop merge, loop interchange and loop collapse. After describing the new notation and transformations, the use of these transformations will be demonstrated by their application on a real-world code example. This example will show the power of the usage of forelem loop nests and the described transformations.

We believe that this methodology might be a solution to the “impedance mismatch” in optimization as described in the first chapter [25]. The mismatch illuminates the fact that there is a mismatch in how application codes are optimized compared to how database statements are optimized. We are convinced that forelem loop nests eliminate this mismatch, even though it is claimed that explicit looping structures give a particular implementation of the query that limits the range of transformations and evaluation choices [65]. Maier says that in databases, iteration is encapsulated so that the system can pick the iteration form. Despite that forelem loop nests appear to be explicit looping structures, the power is in the usage of “index sets”. Because of the use of “index sets”, iteration is still encapsulated and the iteration form is picked during optimization. Which iterations and which index sets are needed will emerge from the optimization phase performed by the code optimization backend. Based on this information, the most efficient way to compute and use the index set is determined, which is equivalent to picking the iteration form.
3.1 The forelem Loop Nest

In this section the basics of forelem loop nests are introduced. Each forelem loop
iterates a (multi)set of tuples. Tuples in these multisets are accessible with sub-
scripts, like ordinary arrays. The subscripts that are accessed through an “index
set” that is associated with the multiset.

The forelem loop will be described through the use of simple SQL queries. Through-
out the discussion, we will use queries inspired by the “Sailors” database
described in [82]. This database consists out of the following table schemas:

Sailors : sid, sname, rating, age
Boats : bid, bname, color
Reserves : sid, bid, day

Let us consider a first query:

```
SELECT S.sname
FROM Sailors S
WHERE S.rating = 7
```

The query is expressed in relational algebra as \(\pi_{sname}(\sigma_{rating=7}(S))\) and is exe-
cuted by performing the selection \(rating = 7\) on table \(S\) (Sailors) and storing
\(sname\) from matching tuples into the result set. A C code to evaluate this query
could look as follows:

```
for (i = 0; i < len(Sailors); i++)
{
   if (Sailors[i].rating == 7)
      add_to_result(Sailors[i].sname)
}
```

In this code fragment, the for loop iterates the full \(S\) table, the if-statement selects
matching tuples, corresponding to the \(\sigma\) operator, and the add_to_result func-
tion performs the task of the \(\pi\) operator.

The main problem with this code fragment is that the looping structure is ex-
licit, which already gives a particular implementation of the query and this limits
the range of transformations and evaluation choices [65]. It is apparent that a full
iteration over the Sailors table is to be done, to check the rating of each Sailor.
This explicit looping structure excludes the possibility to, for example, exploit an
index on the rating values. Additionally, more complex query constructs, such
as distinct and group by, require more complicated code to represent using only
standard C language constructs such as for and if. The downside of this is that
more complicated code is harder to apply transformations to and hides the actual
problem at hand.

Ideally, only those rows are iterated for which the condition \(rating = 7\) holds
true. This is similar to what an index on the column rating would accomplish.
One way to accomplish this is to move the definition of these conditions into the
loop control structure, in our case the for statement. As a result, the explicit if
statements are eliminated, which paves the way for the application of a larger range of optimizations. The above query loop written using `forelem` looks like the following:

\[
\text{forelem} (i; i \in \text{pS.rating}[7]) \\
R = R \cup (S[i].\text{sname})
\]

This code fragment is read as follows: with \(i\), iterate over each index into table \(S\) for which \(\text{rating} == 7\) holds true. For these \(i\), we append a tuple containing the value of \(\text{sname}\) for index \(i\) into table \(S\) to the result set \(R\).

Even though the `forelem` loop appears to be very similar to a `foreach` loop that exists in many common programming languages, there is one distinguishing feature. This concerns the notation \(\text{pS.rating}[7]\). This denotes that a set of indices into table \(S\) will be returned for which the `rating` field equals 7. This is similar to an index set as is commonly used in DBMSs, and we will also use this term to refer to the sets of indices we define here. The fact that an index set contains indices is indicated by the prefix \(p\), from pointer. Note that the order in which the indices appear in the index set is not defined. From this follows that the exact semantics of how the table \(S\) will be iterated are not set in stone at this point. Contrary to the original for loop, the `forelem` loop does not have an explicit looping structure and does not impose a particular implementation of the query. Because of this, we are not limited in the range of transformations and iteration schemes we can apply. Index sets are the essence of `forelem` loop nests as they encapsulate iteration and simplify the query loop code so that aggressive compiler optimizations can be successfully applied.

Before proceeding, some further notation and terminology is introduced first. In this chapter, the focus is on expression SQL queries as `forelem` loops. As a result, the `forelem` loops will iterate database tables. A database table contains tuples, that contain data for one or more columns. In a database table a tuple is not necessarily unique, therefore a database table is a multiset. A multiset is a set in which elements may occur more than once, furthermore, the order of items in the multiset is irrelevant. So, in general, `forelem` loops specify iteration of (a subset of) a multiset of tuples.

Let \(D\) be a multiset representing a database table. \(D\) can be indexed with a subscript \(i\) to get access to a tuple, or row, in \(D\): \(D[i]\). A specific field of a row can be accessed with \(D[i].\text{field}\) where \(\text{field}\) is a valid field of \(D\). Without subscript, an entire column is selected resulting in a multiset containing all values of that column: \(D.\text{field}\).

In `forelem` codes that have been generated from SQL statements, the loop body often outputs tuples to a temporary or result set. Temporary sets are generally named \(T_1, T_2, ..., T_n\) and result sets (or output relations) \(R_1, R_2, ..., R_n\). These temporary tables and result sets are both multisets. The semantics that apply to multisets representing database tables apply to temporary tables as well.

An index set is a set containing subscripts \(i \in \mathbb{N}\) into an array. Since each array subscript is typically processed once per iteration of the array, these subscripts are stored in a regular set. Index sets are named after the array they refer to, prefixed with “p”.
3.1. The forelem Loop Nest

\( pD \) represents the index set of all subscripts into a database table \( D \): \( \forall t \in D : \exists i \in pD : D[i] = t \). \( D \) can also be a temporary table \( \mathcal{T} \). All rows of \( D \) are visited if all members of \( pD \) have been used to subscript \( D \). Random access by subscript into \( pD \) is not possible, instead all accesses are done using the \( \in \) operator. \( i \in pD \) stores the current index into \( i \) and advances \( pD \) to the next entry in the index set.

The part of the table that is selected using an index set can be narrowed down by specifying conditions. For example, the index set denoted by \( pD.field[k] \) returns only those subscripts into \( D \) for which \( field \) has value \( k \). This can also be expressed as follows:

\[
\text{pD.field}[k] \equiv \{ i \mid i \in pD \land D[i].field == k \}
\]

Similarly, the index set from the example query can be expressed as follows:

\[
\text{pS.rating}[7] \equiv \{ i \mid i \in pS \land S[i].rating == 7 \}
\]

When a match on multiple fields is required, the single column name is replaced with a tuple of column names:

\[
\text{pD.(field1,field2)}[(k_1,k_2)] \equiv \{ i \mid i \in pD \land D[i].field_1 == k_1 \land D[i].field_2 == k_2 \}
\]

Instead of a constant value, the values \( k_n \) can also be a reference to a value from another table. To use such a reference, the table, subscript into the table and field name must be specified, e.g.: \( D[i].field \). This notation is especially suited for expressing equi-joins.

**SELECT** S.sname  
**FROM** Sailors S, Reserves R  
**WHERE** S.sid = R.sid AND R.bid = 103

This query finds the name of all sailors who have reserved the boat with id 103. It contains an equi-join on the sid fields from the Sailors and Reserves tables. Expressed in forelem, we obtain:

\[
\text{forelem} \ (i; \ i \in pR.bid[103])
\]

\[
\text{forelem} \ (j; \ j \in pS.sid[R[i].sid])
\]

\( R = R \cup (S[j].sname) \)

If a value should not be tested for equality but rather for greater or less than, a different notation is used. Instead of a single value an interval is written:

\[
\text{pD.field}[(\infty,k)] \equiv \{ i \mid i \in pD \land D[i].field < k \}
\]

Even though the notation implies a single index set, the interval is represented as the union of the individual index sets:

\[
\text{pD.field}[(\infty,k)] \equiv \bigcup_{i=-\infty}^{k-1} \text{pD.field}[i]
\]

Note that when dealing with bounds of infinity, iterating over each possible index set with \( field = i \) is not useful if there are no subscripts into the table for a
Regular set operations on index sets are possible, but only on index sets that relate to the same database table. Index sets contain subscripts into a specific table and therefore it is not possible to, for example, union index sets relating to different tables. Often, the union operation will be of use. For example, given two index sets on a different value of a field, say “color”, the union of these index sets yields all subscripts into the database table for which color is either value.

3.2 Relationship Between SQL and forelem

In this section, we briefly sketch how SQL statements are translated into forelem loop nests. SQL statements with an arbitrary number of joins can be written as a forelem loop nest while preserving correctness of the results. This is because both the SQL statement and the corresponding forelem loop nest set up the same Cartesian product. This fact will be used to reason about the correctness of the translation and the conditions under which transformations can be applied on the forelem loops.

Consider the following query performing a join:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```

which is expressed in relational algebra as \( \pi_{\text{sname}}(\sigma_{S.sid=R.sid \land R.bid=103}(S \times R)) \), or more commonly using the join operator: \( \pi_{\text{sname}}(\sigma_{R.bid=103}(S \bowtie R)) \). Theoretically, a join is performed by first setting up the Cartesian product over \( S \) and \( R \) and secondly selecting tuples which match the given conditions. We can write the first relational algebra expression as a forelem loop nest. The part \( S \times R \) can be written as follows:

\[
\text{forelem } (i; i \in pS) \\
\text{forelem } (j; j \in pR) \\
S_1 \quad R = R \cup (S[i].*, R[j].*)
\]

where \( S[i].* \) denotes all fields of table \( S \) at subscript \( i \). In the result tuple all fields of \( S \) are suffixed with \( S \). This loop nest sets up the Cartesian product \( S \times R \) at statement \( S_1 \), which stores the Cartesian product in \( R \). After executing the loop nest, \( R \) is equivalent to what would be produced by the relational algebra expression \( S \times R \).

The selection operator \( \sigma \) is implemented by making a pass over the result table and only storing matching tuples in a new result table. Of the matching tuples we only store the requested fields to implement the \( \pi \) operator.

\[
\text{forelem } (i; i \in pR) \\
\quad \text{if } (R[i].sid^S = R[i].sid^R \land R[i].bid^R = 103) \\
S_1 \quad R_2 = R_2 \cup (R[i].sname^S)
\]
3.3 Transformations on forelem Loop Nests

By application of the Temporary Table Reduction transformation that will be described in Section 4.4.2, both loops can be merged into one:

```
forelem (i; i ∈ pS)
forelem (j; j ∈ pR)
S1  if (S[i].sid == R[j].sid &amp; R[j].bid == 103)
S2   R = R ∪ (S[i].sname)
```

In general, we say that a perfectly nested forelem loop nest of the following form:

```
forelem (i1; i1 ∈ pT1)
forelem (i2; i2 ∈ pT2)
...
forelem (in; in ∈ pTn)
S1  R = R ∪ (T1[i1].field, ..., Tn[in].field)
```

sets up a Cartesian product of the tables T1, T2, ..., Tn at statement S1. The Cartesian product, or rather the part of the Cartesian product that is accessed, must be preserved under any transformation for the query to yield correct results.

### 3.3 Transformations on forelem Loop Nests

forelem loop nests were devised such that common loop transformations could be applied to the resulting code. A number of transformations that can be applied to forelem loop nests are discussed in this section. These transformations are based on existing optimizing compiler techniques, such as these discussed in Chapter 1 (Section 1.1), and have been tailored for usage with forelem loop nests.

Generally, compiler optimizations are governed by data dependence analysis [58, 3, 5, 104]. The analysis results in data-dependence relations which reflect the constraints on the statement execution order. These constraints determine whether a given transformation can be applied without affecting the correctness of a program.

A related analysis is def-use analysis [2, 50]. In this analysis statements are analyzed to see whether they are a definition (an assignment) or a use of a value. This analysis is used to find unused variables, or to infer the current value of a variable by looking at preceding definitions of the variable in the def-use chain.

#### 3.3.1 Loop Invariant Code Motion

Loop Invariant Code Motion is a kind of common subexpression elimination where statements which are invariant under the loop’s iteration variable can be moved to an outer loop or completely out of the loop nest. Within the forelem framework this transformation is generally used to move condition testing of array fields to outer loops to prune the iteration space, or to inner loops so that further loop transformations will be enabled. For example:

```
forelem (i; i ∈ pX)
forelem (j; j ∈ pY)
```
if (X[i].field2 == value && Y[j].field2 == X[i].field1) 
    \( R = R \cup (Y[j].field1) \)

compares the value \( X[i].field2 \) with a constant value. The reference \( X[i].field2 \) is invariant under the inner loop, so can be moved to the outer loop. Fully moving the condition test out of the loop nest is not possible, because the array reference is variant under the outermost loop. The result is:

\[
\text{forelem} (i; i \in pX) \\
\text{if} (X[i].field2 == value) \\
\text{forelem} (j; j \in pY) \\
\text{if} (Y[j].field2 == X[i].field1) \\
\quad \quad \quad R = R \cup (Y[j].field1)
\]

Similarly, statements can be moved to the innermost loop, to enable the application of loop transformations, such as Loop Interchange.

### 3.3.2 Loop Interchange

The Loop Interchange transformation is derived from the common loop interchange transformation applied by optimizing compilers discussed in Chapter 1 and reorders the nesting of loops in a loop nest. To perform this transformation, Loop Invariant Code Motion is used to move the conditions to the inner loop before the loop nest is reordered and back to the outermost loop after the reordering.

The standard Loop Interchange transformation changes the order in which the statements in the loop are executed. This transformation is only valid if the new execution order preserves all dependencies of the original execution order [104]. Commonly, data-dependence analysis [58, 3, 5] is employed to formally verify whether the data-dependence relations are preserved across loop transformations. In general, only certain loop-carried dependencies can prevent application of Loop Interchange. A forelem loop does not specify a particular execution order and therefore loop-carried dependencies cannot exist. As a consequence, interchanges of loops in a perfect loop nest are always valid.

Loop-carried dependencies are only caused by dependencies of the loop bounds of inner loops on outer loop iteration counters. In this case, Loop Invariant Code Motion is first used to move the conditions to the inner loop before the loop nest is reordered and back to the outermost loop after the reordering. This way, Loop Interchange is applied to a perfectly nested loop nest.

Within the forelem framework the Loop Interchange transformation is used to reorder loops such that as many conditions as possible are tested in the outermost loop to prune the search space. As an example, consider the following loop nest over tables \( X \) and \( Y \) with a result table \( R \):

\[
\text{forelem} (j; j \in pY) \\
\text{forelem} (i; i \in pX.\text{field1, field2})[(Y[j].field2, value))] \\
\quad \quad \quad R = R \cup (Y[j].field1)
\]

First, the conditions of all loops are written as if-statement and moved to the inner loop nest using Loop Invariant Code Motion:
forelem (j; j ∈ pY)
  forelem (i; i ∈ pX)
    if (X[i].field1 == Y[j].field2 && X[i].field1 == value)
      $R = R \cup \{Y[j].field1\}$

After the preparatory step, the forelem loop nest is in the perfectly nested form and generates the cross product of tables X and Y at the if-statement. The only dependency in this loop is the true dependency on $R$ in consecutive iterations. On $R$ the $\cup$ operator is used to indicate that tuples are being added to a (result) set. In this code fragment, no sort order is imposed on the result set, so the order in which the tuples appear in $R$ does not matter (as long as the tuples are correct). This breaks the true dependency. Given that there are no other dependencies, the loops can be interchanged freely which does not change the contents of the generated cross product:

forelem (i; i ∈ pX)
  forelem (j; j ∈ pY)
    if (X[i].field2 == value)
      forelem (j; j ∈ pY)
        if (Y[j].field2 == X[i].field1)
          $R = R \cup \{Y[j].field1\}$

Next, Loop Invariant Code Motion is applied to move conditions to the outermost loops:

forelem (i; i ∈ pX)
  if (X[i].field2 == value)
    forelem (j; j ∈ pY)
      if (Y[j].field2 == X[i].field1)
        $R = R \cup \{Y[j].field1\}$

As a final step, the conditions are moved from the if-statements into the index sets in the forelem loop nest:

forelem (i; i ∈ pX.field2[value])
  forelem (j; j ∈ pY.field2[X[i].field1])
    $R = R \cup \{Y[j].field1\}$

Note, that as a result of this transformation, the comparison with value is performed in the outermost loop, which effectively prunes the iteration space.

### 3.3.3 Loop Fusion

Loop Fusion [52] is a traditional compiler optimization that can be readily applied to forelem loops. The transformation can, under certain conditions, merge two loops (at the same level if contained in a larger loop nest) into a single loop. Application of Loop Fusion is only prohibited by certain loop-carried dependencies. Such loop-carried dependencies do not exist in forelem loops. Therefore, Loop Fusion can be applied on two adjacent forelem loops if the iteration spaces of the two loops are equal. This is the case if the index sets for both loops refer to the same table and contain the same set of subscripts into these tables. After Loop Fusion has been applied, the bodies of both loops are then executed for the same set of subscripts into the same array. For example:
forelem (i; i ∈ pTable1)
   \( R_1 = R_1 \cup (Table1[i].field1) \)
forelem (i; i ∈ pTable1)
   \( R_2 = R_2 \cup (Table1[i].field2) \)

can be rewritten into the following:

forelem (i; i ∈ pTable1)
   {   \( R_1 = R_1 \cup (Table1[i].field1) \)
       \( R_2 = R_2 \cup (Table1[i].field2) \)
   }

Note that *forelem* loops generally only access the array being iterated using the subscript of the current iteration. E.g., an access into an array always has the form \( i \) and not \( i + 2 \) or similar. As a consequence, a condition preventing Loop Fusion from being applied will in general not occur.

### 3.3.4 Loop Merge

A typical query in a database application has the following structure when expressed using *forelem* loops:

forelem (i; i ∈ pS.rating[7])
   \( R = R \cup (S[i].sname) \)
while (row ∈ R)
   print(row.sname);

The *forelem* loop is a loop producing result tuples, the *while* loop is a loop consuming these result tuples. Both loops enumerate the same iteration space, namely the set of all matching tuples. In this particular example, the temporary storage \( R \) is not required and can be eliminated. To eliminate the usage of temporary storage, the loops have to be merged, similar to how loops enumerating the same iteration space can be fused using Loop Fusion under certain conditions:

forelem (i; i ∈ pS.rating[7])
   {   row = S[i].sname;
       print(row.sname);
   }

We will refer to this transformation as the Loop Merge transformation. Note that as a further optimization, the references to the result row can be rewritten to be references immediately into the database table.

Sometimes a preparatory transformation has to be done on the code before it is possible to perform the Loop Merge transformation. Consider the following code fragment:
3.3. Transformations on *forelem* Loop Nests

3.3.1 Loop Merge

To make it possible to apply the Loop Merge transformation, the else-clause has to be eliminated. We can reason that when the if-condition is true, the body of the while loop will never be executed. The is_empty function that is called in the if-condition is under our control and we can ascertain that this will not introduce side effects. This also holds true for the condition in the while statement. We can now place the while loop before the if-statement and eliminate the else clause:

```plaintext
forelem (i; i ∈ pS.rating[7])
R = R ∪ (S[i].sname)
if (is_empty(R))
    print("There are no matches.");
else
    while (row ∈ R)
        print(row.sname);
}
```

As a result, it is possible to apply Loop Merge.

More preparatory transformations are required in order to cover a wide range of database applications. Within the limitations of this thesis, they will not further be discussed.

3.3.5 Loop Collapse

With the Loop Collapse transformation, two *forelem* loops are collapsed into a single loop. In conjunction with this, the two tables iterated by these loops are merged into a new table. This is inspired by the original Loop Collapse transformation used in optimizing compilers which rewrites two levels of loops as one level of loop by using a one-dimensional representation of two-dimensional arrays [100]. Loop Collapse is a vectorization transformation and can only be applied on serial loops, that is, no loop-carried dependencies are present.

An sample scenario where Loop Collapse might pay off is when a second query is executed for each result tuple generated by the main query. In terms of the *forelem* intermediate representation, this second query is executed from within the loop body of the main query, as will be demonstrated in an example. After the transformation both the main and second queries can be satisfied by an iteration over a single table. Secondly, this transformation can be used to search for a potentially more effective table layout to compute the given query.

Consider the following loop nest over tables X and Y with a result table R:
forelem (i; i \in pX)
    forelem (j; j \in pY.field2[X[i].field1])
    \mathcal{R} = \mathcal{R} \cup (Y[j].field1)

Similar to the Loop Interchange transformation, we first write the conditions of all loops as \textit{if}-statements and move these into the inner loop using Loop Invariant Code Motion:

forelem (i; i \in pX)
    forelem (j; j \in pY)
    \text{if} (Y[j].field2 == X[i].field1)
    \mathcal{R} = \mathcal{R} \cup (Y[j].field1)

We note that the \textit{forelem} loop nest is now in the perfectly nested form and generates the cross product of tables \textit{X} and \textit{Y} at the \textit{if}-statement. The loop is serial in that there are no loop-carried dependencies and due to the use of the \textit{\cup} operator and the fact that no sort order is imposed no dependencies are enforced on \mathcal{R}.

The two loops are now collapsed as follows. First, a new table \textit{X} \times \textit{Y} is created by taking the cross product of \textit{X} and \textit{Y}. Secondly, the two loops are replaced with a single loop which iterates over the new table. The conditions and code to append a result are rewritten to use the new table.

forelem (k; k \in pX \times Y)
    \text{if} (X \times Y[k].field2 == X \times Y[k].field1^Y)
    \mathcal{R} = \mathcal{R} \cup (X \times Y[k].field1^Y)

The cross product \textit{X} \times \textit{Y} that has been created in the course of the transformation is equal to the cross product that is generated by the relational algebra expression \textit{X} \times \textit{Y}. Hence the \textit{forelem} loop iterates over the same relation as the selection operator. Exactly the same cross product is generated at the \textit{if}-statement compared to the non-collapse loops.

At last, conditions are moved into the \textit{forelem} statement again:

forelem (k; k \in pX \times Y\\cdot field2\textsuperscript{Y}[X \times Y\cdot field1\textsuperscript{X}])
    \mathcal{R} = \mathcal{R} \cup (X \times Y[k].field1^Y)

In this step, new syntax is introduced. Note that the value to match for the condition on \textit{field2}^\textit{Y} is \textit{field1}^\textit{X}. When no table and no subscript specifying a row in the table are explicitly given, then the same table and row are used from which the field that is being compared to is obtained. In this case, we are comparing the value to field \textit{field2}^\textit{Y} in table \textit{pX} \times \textit{Y}, so \textit{field1}^\textit{X} is taken from table \textit{pX} \times \textit{Y} as well. Both of the fields are retrieved from the same row, in this case both from the row at subscript \textit{k}.

So far, we have considered a perfectly nested loop. With additional analysis it is also possible to transform imperfect loop nests into a form such that Loop Collapse can be applied. Let us consider the following loop nest, for which we in light of this discussion assume that \mathcal{R}_2 is never empty at statement \textit{S}_4:
3.3. Transformations on *forelem* Loop Nests

```plaintext
forelem (i; i ∈ pItems)
{
    userId = Items[i].userId
S_1
    R_2 = ∅
forelem (j; j ∈ pUsers.id[userId])
S_3
    R_2 = R_2 ∪ (Users[j].name)
S_4
    R = R ∪ (r_2 ∈ R_2)
}
```

This construction was inspired by a case from the RUBiS benchmark [75], which is further discussed in Section 3.4. Note that statement S_4 fetches a single tuple from R_2 which is subsequently stored in result set R.

The loop nest differs from the perfectly nested loop nest because the main result R is assigned at S_4, which is outside of the inner loop body and secondly because a second result set R_2 is used. At S_2 the entire Cartesian product is generated by the loop nest, however, the full product is not stored in a result set. At S_3, only fields are stored from tuples where Users.id and Item.userId match. Due to S_4, only the first of these matches makes its way into the result set R.

We can detect this more formally by applying def-use analysis on the loop nest. We assume a modified form of def-use analysis, which can interpret *forelem* loops and its index sets. Def-use analysis will indicate that after running the inner loop R_2 will either contain ∅ (which we will in light of this discussion ignore), or one or more user names generated by S_3. Additionally, we know from the analysis that only the first item of R_2 is used (S_4). For the inner loop this means it is sufficient to only perform the first iteration and to eliminate the remainder of the loop:

```plaintext
forelem (i; i ∈ pItems)
{
    userId = Items[i].userId
    j = j ∈ pUsers.id[userId]
S_4
    R = R ∪ (Users[j].name)
}
```

In this loop, the execution flow of the original loop becomes more obvious. For each item, the username of the first user which matches the Item’s userId is fetched.

With this knowledge, we will now transform the loop nest into a form such that Loop Collapse can be applied. Given a table Users with fields id and name, we add a column idMask such that idMask is set to 1 for each occurrence of a row in Users that makes it into R. In this case, idMask is 1 for each first occurrence of a value in id and 0 for each following occurrence of that value.

Observe that when the column id solely contains unique values, or when it has been marked as a key in the original table schema, all values in idMask are 1. When we use the new column in the loop nest, we obtain:

```plaintext
forelem (i; i ∈ pItems)
{
S_1
    R_2 = ∅
```
Due to the use of the mask, we are now assured that $S_3$ will only generate a single tuple for a matching user. We can merge statements $S_3$ and $S_4$ and eliminate the usage of $R_2$:

$$forelem \ (j; \ j \in \ p\text{Users} \cdot (\text{id}, \text{idMask})[(\text{userId}, 1)])$$

$$S_3 \ \ R_2 = R_2 \cup (\text{Users}[j].\text{name})$$

$$S_4 \ \ R = R \cup (r_2 \in R_2)$$

The loop is now in perfectly nested form and suitable to be transformed using the defined transformations.

### 3.3.6 Reverse Loop Collapse

Reverse Loop Collapse is exactly the reverse operation of the Loop Collapse transformation. Of a specified $forelem$ loop, the accessed table is split into two tables and the loop is replaced with two loops, each iterating one of the two newly created tables. How the fields of the large table will be divided over the two new tables, or how the table will be split, has to be specified. This information will be giving by the code optimization backend that is driving the application of the different transformations.

The Reverse Loop Collapse transformation can be applied after application of several Loop Collapse and Loop Interchange transformations. Due to reorderings of the loops in the loop nest, it is likely that different tables will be chosen to be split, compared to the tables which were combined (collapsed). This will potentially lead to different schemas for the database tables which allow the query to be computed more efficiently.

During optimization, the code optimization backend plays a central role in determining how loop nests are reordered, split and combined. When guiding the optimization, the backend will take into account all queries of a given application code. The intention is to arrive at database table schemas which make the application code as a whole run more efficiently and not to optimize for one specific query of that application code.

### 3.3.7 Horizontal Iteration Space Reduction

The aim of horizontal iteration space reduction is to remove unused fields from a table’s schema. Let $T$ be a table with fields $\text{field1}$, $\text{field2}$, $\text{field3}$ and $\text{field4}$, $C$ a list of condition fields $C \subseteq (\text{field1} \ \text{field2})$ and $V$ a list of values. Consider the loop nest:

$$forelem \ (k; \ k \in pT.C[V])$$

$$R = R \cup T[k].\text{field1} + T[k].\text{field2}$$
We define a new table $T' \subseteq T$ with fields $\text{field1}$, $\text{field2}$ and replace the use of $T$ with $T'$ in the loop:

$$\text{forelem } (k; \ k \in pT'.C[V])$$

$$R = R \cup T'[k].\text{field1} + T'[k].\text{field2}$$

The loop now iterates the table $T'$ which does not contain the fields $\text{field3}$ and $\text{field4}$.

### 3.3.8 Vertical Iteration Space Reduction

The Vertical Iteration Space Reduction transformation is primarily used after the application of the Loop Collapse transformation. Recall that Loop Collapse may introduce conditions that test whether two fields of the same row are equal. In Vertical Iteration Space Reduction, rows for which such conditions do not hold are eliminated from the table. This elimination is valid because these rows are never visited in the inner loop. Two different cases are distinguished: $T.\text{field1}[\text{field2}]$ and $T.\text{field}[k]$.

#### Case 1. $T.\text{field1}[\text{field2}]$

On a table $T$, let the condition be $T.\text{field1}[\text{field2}]$. This notation implies table $T$ contains both fields name $\text{field1}$ and $\text{field2}$. All rows for which $T.\text{field1} \neq T.\text{field2}$ holds true are removed from the table. In the resulting table $T'$, $\forall t \in T'$: $t.\text{field1} = t.\text{field2}$, so that either $\text{field1}$ or $\text{field2}$ can be removed. Let $T$ be a table with fields $\text{field1} ... \text{fieldn}$ and consider the loop nest:

$$\text{forelem } (k; \ k \in pT.\text{field1}[\text{field2}])$$

$$R = R \cup T[k].\text{field3}$$

A new table $T'$, with all fields of $T$ except $\text{field2}$, is defined as follows:

$$T' = \{ t | t \in T \land t.\text{field1} = t.\text{field2} \}$$

and replaces $T$ in the loop nest, additionally all uses of $\text{field2}$ are replaced with $\text{field1}$:

$$\text{forelem } (k; \ k \in pT')$$

$$R = R \cup T'[k].\text{field3}$$

Note that in this specific case $\text{field1}$ is no longer used after the transformation and can be removed by applying horizontal iteration space reduction.

#### Case 2. $T.\text{field}[k]$

Case 2 is similar to Case 1, however since we do not reduce an equality between two fields in Case 2, no field is removed by the reduction operation.

On a table $T$, let the condition be $T.\text{field}[k]$ All rows for which $T.\text{field} \neq k$ holds true are removed from the table. In the resulting table $T'$, $\forall t \in T'$: $t.\text{field} = k$.

Let $T$ be a table with fields $\text{field1} ... \text{fieldn}$ and consider the loop nest:
forelem \((i; \ i \in \text{pT.field}[k])\)
\[
\mathcal{R} = \mathcal{R} \cup T[i].field3
\]

A new table \(T'\) is defined with the same fields as \(T\) as follows:
\[
T' = \{t \mid t \in T \land t.field = k\}
\]

Note that in this specific case \text{field} is no longer used after the transformation and can be removed by applying horizontal iteration space reduction.

This reduction can also be applied if instead of a constant \(k\), an interval is given. All rows for which the complement of the interval holds true are removed from the table. For example, for an interval \((-\infty, 10)\) all rows for which the field matches the complemented interval \([10, \infty)\) are removed from the table. This results in the table \(T'\) with \(\forall t \in T': t.field < 10\).

### 3.3.9 Table Reduction Operators

In many database operations first an expanded result table is generated after which it is reduced to using just a single subset of columns and/or rows. More specifically, this happens when explicit Cartesian products are computed to perform table joins, for instance in the Loop Collapse transformation. Note that in our transformation framework, we have implicit transformations which immediately reduce the generated Cartesian product to a set of reasonable size. These reductions are defined as Horizontal and Vertical Iteration Space Reduction and are driven by the conditions and selected columns of the query. In fact, because the Cartesian product only is referred to in intermediate codes when transformations are taking place, the full product is usually never instantiated.

We also define explicit table reduction operators in our framework. Whereas implicit table reduction operators do not affect the final result set of a query, explicit table reduction operators do. The expansion of queries in our framework into loop-based programs potentially enables the use of def-use analysis to detect explicit reductions of these expanded tables later on in the code. The following program fragment demonstrates two examples of such explicit reductions, one column based and one row based:

forelem \((i; \ i \in \text{pS.rating}[7])\)
\[
\mathcal{R} = \mathcal{R} \cup (S[i].*)
\]

while (row \in \mathcal{R})
\[
\text{if (row.age > 18)}
\]
\[
\text{print(row.sname);}
\]

We consider the program fragment before applying Loop Merge, so that the individual loops performing the query \(\text{SELECT * FROM Sailors WHERE rating = 7}\) and the loops processing the result set are clearly visible. Firstly, we observe that only rows with \(\text{age} > 18\) are being further processed and other rows are discarded. This allows us to apply row-based explicit table reduction to eliminate all rows with \(\text{age} \leq 18\) from the result set \(\mathcal{R}\). Secondly, we observe that the field \text{sid} is not used in the program fragment, although it is fetched from the table due to the * operator. Using column-based explicit table reduction, we further reduce the size of the result set by no longer storing \text{sid} in result rows.
3.3.10 Combined transformations

The real power of the transformations presented in this section is unleashed when they are combined. Let us consider the following code fragment, which is generated from a simple database application code performing the query

```
SELECT *
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```

and enumerating the result set to output the sname field for each row. This results in the following intermediate code:

```c
forelem (i; i ∈ pR.bid[103])
  forelem (j; j ∈ pS.sid[R[i].sid])
    R = R ∪ (S[j].sid, S[j].sname, S[j].rating, S[j].age,
             R[i].sid, R[i].bid, R[i].day)
  while (row ∈ R)
    print(row.sname);
```

As a first transformation, we apply Loop Collapse to the loop nest. This transformation will also rewrite all references to columns in the program:

```c
forelem (i; i ∈ pR×S.(bidR,sidS)[(103,sidR)])
  R = R ∪ (R×S[i].sidS, R×S[i].snameS, R×S[i].ratingS,
  while (row ∈ R)
    print(row.sname);
```

Secondly, we perform a Loop Merge transformation to merge the `while` loop consuming the tuples with the `forelem` loop which produces these.

```c
forelem (i; i ∈ pR×S.(bidR,sidS)[(103,sidR)])
{
  row = (R×S[i].sidS, R×S[i].snameS, R×S[i].ratingS, R×S[i].ageS,
         R×S[i].sidR, R×S[i].bidR, R×S[i].dayR)
  print(row.sname);
}
```

We observe that of all columns added to each result tuple, only the `sname` column is used. Using the explicit table reduction operator, we reduce the size of the generated result set.

```c
forelem (i; i ∈ pR×S.(bidR,sidS)[(103,sidR)])
{
  row = (R×S[i].snameS)
  print(row.sname);
}
```
As a last step, the assignment to row could be eliminating by having the print statement immediately access the database table.

These transformations drastically reduce the amount of overhead in the database program, that is usually imposed by the API provided by DBMS interfacing libraries. As has been shown in Chapter 2 this overhead can be considerable and removal leads to a further reduction in non-essential instructions executed by a database program. In some cases, this is another factor of 2 reduction on top of the instruction reduction achieved by releasing the dependency on a stand-alone DBMS.

### 3.4 Example Application of the Transformations

In this section we study an example application of the forelem loop and transformations described in this chapter to a code fragment from the RUBiS [75] benchmark. We have performed many of these transformations on the RUBiS benchmark in order to estimate the amount of non-essential instructions performed by web applications shown in Chapter 2. We found that approximately 90% of the executed instructions were non-essential and could thus be eliminated. Elimination of these non-essential instructions had an immediate impact on performance. We showed that because of these reductions in instruction count and execution time, the elimination of non-essential instructions is expected to significantly reduce energy use of server systems running web applications by 70% to 90%.

The following code fragment, written in pseudocode similar to PHP and edited for clarity, is based on the file ViewUserInfo.php from the RUBiS benchmark [75]:

```php
$commentsResult = mysql_query("SELECT * FROM comments WHERE " . 
    "comments.to_user_id=$userId");
if (mysql_num_rows($commentsResult) == 0)
    print("<h2>There is no comment for this user.</h2><br>

else
{
    print("<DL>
    while ($commentsRow = mysql_fetch_array($commentsResult))
    {
        $authorId = $commentsRow["from_user_id"];
        $authorResult = mysql_query("SELECT nickname FROM users " . 
            "WHERE users.id=$authorId");
        $authorRow = mysql_fetch_array($authorResult);
        $authorName = $authorRow["nickname"];
        $date = $commentsRow["date"];
        $comment = $commentsRow["comment"];  
        print("<DT><b><BIG><a href="/PHP/ViewUserInfo.php?userId=".$authorId.
```
When the SQL queries that are performed by calling the DBMS API are replaced with `forelem` loop nests which execute in this process, we obtain:

```php
forelem (i; i ∈ pComments.to_user_id[$userId])
R₁ = R₁ ∪ (comments[i].id, comments[i].from_user_id,
            comments[i].to_user_id, comments[i].item_id,
            comments[i].rating, comments[i].date,
            comments[i].comment)
if (is_empty(R₁))
    print("<h2>There is no comment for this user.</h2><br>
else
    {
        print("<DL><n>"
        while ($commentsRow ∈ R₁)
        {
            $authorId = $commentsRow["from_user_id"];

            forelem (j; j ∈ pUsers.id[$authorId])
            R₂ = R₂ ∪ (users[j].nickname)

            $authorRow = r₂ ∈ R₂;
            $authorName = $authorRow["nickname"];

            $date = $commentsRow["date"];
            $comment = $commentsRow["comment"];
            print("<DT><b><BIG>".
                "/a href="/PHP/ViewUserInfo.php?userId=".$authorId.
                "/"$authorName</a>"/BIG></b> wrote the ".$date.
                "/DD><i>".$comment."</i><p>
            }
        print("</DL><n>"
    }
As a first transformation, we will merge the `forelem` loop producing the tuples into result set $R₁$ with the while loop consuming tuples from that result set. To do this, we first have to perform a preparatory transformation, similar to the one outlined in Section 3.3.4. We will move the `if`-statement checking `is_empty` to after the merged loop and change it to check how many result tuples were processed. This is safe, because the true clause of the `if`-statement will only be run if the query loop did not produce (and in the merged case, process) any result tuple. Secondly, the `is_empty` function in the `if`-statement is under our control and we can ascertain that the function does not introduce any side effects.
For the second *forelem* loop, producing into $R_2$, we observe that consistently only the first result of the set is used. This will be caught by def-use analysis, similar to the example described in Section 3.3.5. We will apply a similar transformation here and use an additional mask column to ensure only one table row is processed by the inner loop. Also in this case, we move the code consuming the result tuples into the inner *forelem* loop body. Code accessing *Comments*[i] is moved as well, which is valid considering $i$ is invariant under the inner loop. The increment of *results* can be moved because we know the inner loop will output at least one and at most one tuple.

At the same time we perform a first explicit table reduction and replace the references into result tuples with direct references into the database table, see Section 3.3.10. Applying these transformations results in the following code:

```plaintext
$results = 0;
forelem (i; i ∈ pComments.to_user_id[$userId])
{
  forelem (j; j ∈ pUsers.(id,idMask)((Comments[i].from_user_id, 1)))
  {
    if ($results == 0)
      print("<DL><n");
    $results++;

    $authorName = Users[j]["nickname"];
    $authorId = Comments[i]["from_user_id"];
    $date = Comments[i]["date"];
    $comment = Comments[i]["comment"];
    print("<DT><b><BIG>"
      "<a href="/PHP/ViewUserInfo.php?userId=".$authorId.
      ">".$authorName</a></BIG></b> wrote the ".$date.
      "<DD><i>".$comment."</i></DD><p><n");
  }
}
if ($results == 0)
  print("<h2>There is no comment for this user.</h3><br><n");
else
  print("</DL><n");
```

For the remainder of the discussion, we will focus solely on the *forelem* loops with the other code removed:

```plaintext
$results = 0;
forelem (i; i ∈ pComments.to_user_id[$userId])
{
  forelem (j; j ∈ pUsers.(id,idMask)((Comments[i].from_user_id, 1)))
  {
    $results++;
  }
}
```
In this particular case, it is interesting to perform the Loop Collapse transformation to merge the Comments and Users tables. For the current discussion, we are not concerned with the cost involved to generate this cross product. As indicated in Section 3.3.9, it is likely that the full table will not be generated due to the implicit table reduction operators that will be applied by the framework. After the merge, we can eliminate the inner `forelem` loop over `Users` and satisfy the query with a single pass over a single table.

We observe the loop is perfectly nested and the Loop Collapse transformation can be applied. The collapsed loop nest looks as follows:

```php
$results = 0;
forelem (i; i ∈ pComments×Users.
             (to_user_idComments, id(Users, idMaskUsers)
              [($userId, from_user_idComments, 1)]))
{
    $results++;
}
```

During the Loop Collapse process, all references to fields of the two tables being merged are rewritten to be references to fields of the combined table. After rewriting the references, the code becomes:

```php
$results = 0;
forelem (i; i ∈ pComments×Users.
             (to_user_idComments, id(Users, idMaskUsers)
              [($userId, from_user_idComments, 1)]))
{
    if ($results == 0)
        print("<DL><n");
    $results++;

    $authorName = Comments×Users[i]."nicknameUsers";

    $authorId = Comments×Users[i]."from_user_idComments";
    $date = Comments×Users[i]."dateComments";
    $comment = Comments×Users[i]."commentComments";
    print("<DT><b><BIG><a href="/PHP/ViewUserInfo.php?userId="$authorId."
    "$authorName"></a></BIG></b> wrote the "$date.
    "$comment."</i><p><n";
}
if ($results == 0)
    print("<h2>There is no comment for this user."<br><n");
else
    print("</DL><n");
```

Through the course of this example, we first eliminated all calls to MySQL's interfacing API and replaced these with equivalents performing the requested operation in place. The calls which request MySQL to evaluate a query, in particular
mysql_query, have been replaced with a forelem loop nest computing the query. Next, we applied Loop Merge, to merge the producer and consumer loops of the executed queries. This typically saves a single full iteration of the result set. Explicit table reduction was applied to eliminate reads of columns from the tables which were not used by the code consuming the retrieved data. Finally, a Loop Collapse was performed, such that the requests for data are now served from a single table instead of two separate tables.

The end result is translated to C code and compiled into a final executable using a high-end optimizing compiler. By applying this methodology to the file ViewUserInfo.php, we have been able to eliminate at least 95% of the instructions executed by the PHP script to generate this web page.

### 3.5 Conclusions

In this chapter, we presented a methodology to remove the division of application and DBMS codes, so that applications can be optimized to their fullest potential using optimizing compiler technology. The methodology is centered around the forelem loop nest, which uses index sets to encapsulate iteration and to simplify the query loop nest so that aggressive compiler optimizations can be successfully applied. Because database queries are expressed in terms of an iterative language using forelem loop nests, it allows for the integration of the code performing evaluation of the database query with the application code. The forelem constructs have been designed to integrate in a normal optimizing compiler work flow, such that the existing body of optimizations can be re-used.

It is not our intention to move forelem forward as a new programming paradigm for programming database applications. Rather, we want our methodology to co-exist with existing development environments and frameworks for database application programming. forelem is solely used as an intermediate representation in a code optimization backend that is able to take an existing database or web application and automatically breaks down the layers.

We are fully aware that the material presented in this chapter is just a starting point and lots of future work remains. For a full implementation of the integration of DBMS and application codes using forelem, several implementation issues need to be addressed. These include the storage layer, sorting, dynamic index set generation, etc. We believe that with this methodology it will become possible to eliminate the majority of the software overhead we described in Chapter 2. Given the huge potential to reduce e.g. energy usage, we think it is definitely worth to explore this area.