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 6

Query Optimization Using the Forelem Framework

6.1 Introduction

This chapter explores the optimization of database queries using just simple compiler transformations. This optimization process is carried out by the consecutive application of simple compiler transformations that are expressed in the forelem intermediate representation as a series of forelem loops. So instead of using a traditional query optimizer which optimizes queries that have been expressed into an initial query execution plan, the compiler transformations are the main query optimization transformations. This approach is different from other compiler-based approaches to query optimization, such as [57, 73], that focus on code generation and propose compiler-based techniques for the generation of efficient executable code from algebraic query execution plans.

The optimization methodology that is proposed in this chapter is part of a larger framework for the vertical integration of database applications. Extensive vertical integration is not possible with traditional query optimization techniques, because when code is generated from query evaluation plans and combined with application code, further applicability of compiler transformations is obscured. Therefore, it is important that queries are expressed, optimized and combined with application code in a way that compiler optimizations can still be successfully exploited. The forelem framework provides such a way. Vertical integration will be more thoroughly discussed in Chapter 7.

The techniques described in this chapter build upon the forelem intermediate representation and the transformations introduced in the previous chapters. First, a number of transformations will be introduced that are specific to the optimization of queries expressed in the forelem intermediate representation. Secondly, the application of the transformations will be illustrated. Thirdly, strategies will be discussed for the sequence in which the transformations should be applied, as well as strategies for the generation of efficient (C/C++) code from the optimized
forelem intermediate representation of the query. Finally, using the TPC-H benchmark [91], it is demonstrated that queries optimized using compiler transformations in the forelem framework have a performance that is comparable to that of contemporary database systems that employ traditional query optimization.

6.2 Specific Forelem Transformations for Query Optimization

This section builds upon the forelem loop and transformations introduced in the preceding chapters. In particular, in this chapter use will be made of the Loop Invariant Code Motion, Loop Interchange and Loop Fusion transformations, which will not be reiterated here. For more details, we refer the reader to Section 3.3.

A number of different compiler transformations are specifically introduced for the optimization of queries expressed in the forelem intermediate representation. Although these transformations support optimization of queries within the forelem framework, these transformations can also be applied in general to optimize overall performance. The main contribution of this section is that the majority of the techniques used for query optimization within the forelem framework can be derived from existing optimizing compiler transformations.

6.2.1 Inline

The Inlining transformation inlines a function into its caller. This transformation is commonly used to inline calls to short functions and methods to save the overhead of performing a function call, or to enable further optimization by considering the code of the inlined function in the context of the code that calls this function.

In the forelem framework, all subqueries are initially expressed as separate functions. After inlining a subquery into its caller, the subquery can be considered together with the surrounding loops in the caller. For example, subqueries are often called from a loop nest and after inlining the compiler might detect that the subquery is invariant to the loop body from which it is called. As a result, the Loop Invariant Code Motion transformation will move the subquery out of the loop.

Consider the following subquery and loop:

```plaintext
subquery0()
{
    count = 0;
    forelem (ii; ii ∈ pA.field1[value])
        count++;
    return count;
}

forelem (i; i ∈ pB)
{
    tmp = subquery0()
```
if (B[i].field2 < tmp)
    \( R = R \cup (B[i].field1) \)
}

the subquery is inlined into the caller as follows:

```c
forelem (i; i ∈ pB)
{
    count = 0;
    forelem (ii; ii ∈ pA.field1[value])
    count++;
    tmp = count;
    if (B[i].field2 < tmp)
        \( R = R \cup (B[i].field1) \)
}
```

This has enabled subsequent transformations to be applied. The loop computing
the count variable is invariant to the loop iterating the table B. The subquery was
“uncorrelated”. Because of this, the loop computing count can be moved out of
the loop:

```c
count = 0;
forelem (ii; ii ∈ pA.field1[value])
    count++;
forelem (i; i ∈ pB)
{
    tmp = count;
    if (B[i].field2 < tmp)
        \( R = R \cup (B[i].field1) \)
}
```

### 6.2.2 Iteration Space Expansion

Within the `forelem` framework a transformation known as Iteration Space Expansion
is defined. This transformation is inspired by the Scalar Expansion transformation, which is typically used to enable parallelization of loop nests. There is also a relation with the expansion of the iteration spaces to transform irregular
access patterns into regular ones [95]. This transformation is briefly described in
this subsection, for a more detailed description see Section 12.3.

Iteration Space Expansion expands the iteration space of a `forelem` loop by removing conditions on its index set. For a loop of the form, with SEQ denoting a sequence of statements:

```c
forelem (i; i ∈ pA.field[X])
    SEQ;
```

the following steps are performed:

1. the condition \( A[i].field = X \) is removed, which expands the iteration space so that the entire array \( A \) is visited,
2. scalar expansion is applied on all variables that are written to in the loop body denoted by \textit{SEQ} and references to these variables are subscripted with the value tested in the condition, in this case \texttt{A[i].field},

3. all references to the scalar expanded variables after the loop are rewritten to reference subscript \texttt{X} of the scalar expanded variable.

### 6.2.3 Table Propagation

The Table Propagation transformation is similar to Scalar Propagation that is typically performed by compilers. In Scalar Propagation, the use of variables whose value is known at compile-time is substituted with that value. For example, in:

\begin{verbatim}
int x = 3;
int y = x + 3;
int z = x * 9;
\end{verbatim}

the uses of \texttt{x} can be replaced with the value of \texttt{x}, 3:

\begin{verbatim}
int x = 3;
int y = 3 + 3;
int z = 3 * 9;
\end{verbatim}

In Table Propagation, the use of a temporary table of which the contents are known is replaced with a loop nest that generates the same contents as the temporary table. This eliminates unnecessary copying of data to create the temporary table, but also enables further transformations because the loop nest that generates the contents of the temporary table can now be considered together with the loop nest that iterates the temporary table. For example, consider the following \texttt{forelem} loops:

\begin{verbatim}
forelem (i; i \in pX.field2[value])
\quad \mathcal{T} = \mathcal{T} \cup (X[i].field1)

forelem (i; i \in pY)
  forelem (j; j \in pY.field2[\mathcal{T}[i].field1])
  \quad \mathcal{R} = \mathcal{R} \cup (Y[j].field1)
\end{verbatim}

The first loop generates a table \( \mathcal{T} \), which is iterated by the second loop. The table \( \mathcal{T} \) is being “streamed” between these consecutive loops. Table \( \mathcal{T} \) can be propagated to the second loop nest:

\begin{verbatim}
forelem (i; i \in pX.field2[value])
\quad \mathcal{T} = \mathcal{T} \cup (X[i].field1)

forelem (i; i \in pX.field2[value])
  forelem (j; j \in pY.field2[X[i].field1])
  \quad \mathcal{R} = \mathcal{R} \cup (Y[j].field1)
\end{verbatim}
The result of the first loop, table \( T \), is now unused. Therefore, the first loop may be eliminated by a succeeding compiler transformation, resulting in:

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

This final result gives the impression that a variant of Loop Fusion has been applied. Rather, in the forelem framework this optimization is expressed as two separate transformations: Table Propagation and Dead Code Elimination. Note that this transformation is a generalized form of the Temporary Table Reduction transformation discussed in Section 4.4.2.

### 6.2.4 Dead Code Elimination

Dead Code Elimination removes statements whose results are not used in any subsequent statements. Such statements can be detected using, for example, def-use analysis. In the forelem framework, tables are treated as variables. As a result, statements that generate tables that are unused in the remainder of the forelem representation of the problem will be removed by Dead Code Elimination.

### 6.2.5 Index Extraction

The Index Extraction transformation extracts the use of an index set from a forelem statement. A new loop is created that iterates the index set and fills a temporary table. The index set in the original loop is replaced with an unconditional iteration of this temporary table. This transformation will transform the following loop:

\[
\text{forelem (i; } i \in pTable1.field1[value1]\text{)}
\]
\[
\text{SEQ;}
\]

into:

\[
\text{forelem (i; } i \in pTable1.field1[value1]\text{)}
\]
\[
\mathcal{T} = \mathcal{T} \cup (\ldots)
\]

\[
\text{forelem (i; } i \in p\mathcal{T}\text{)}
\]
\[
\text{SEQ;}
\]

In fact, this transformation can be seen as the opposite of Table Propagation. The Index Extraction transformation extracts one or more forelem loops from a loop nest to a new loop nest that generates a temporary table. The original loop nest is modified to replace the extracted loops with a loop iterating the temporary table. This transformation is useful in the following example:

\[
\text{forelem (i; } i \in pTable1.(field2,field3)[(value1, value2)]\text{)}
\]
\[
\text{forelem (j; } j \in pTable2.field1[Table1[i].field1]\text{)}
\]
\[
\text{forelem (k; } k \in pTable3.field1[Table2[j].field2]\text{)}
\]
\[
\text{forelem (l; } l \in pTable4.field1[Table3[k].field1]\text{)}
\]
\[
\text{if (Table4[l].field2 == value3)}
\]
\[
\text{SEQ;}
\]
where SEQ denotes a sequence of statements. In this case, the transformations decided to move the tests of the conditions on Table1 to the outermost loop, because two conditions are tested and potentially prunes the search space by a large extent. Due to the dependences between the other tables, the condition for Table4 is only tested in the inner loop.

Suppose that Table4 and subscript 1 are not used in SEQ, then the iteration of this array is not necessary in this loop nest. Instead, the subscripts $k$ that should be iterated can be computed before executing this loop nest. This is done by executing the inner two loops and finding all subscripts $k$, for which a subscript 1 exists that satisfies the condition $\text{Table4}[1].\text{field2} == \text{value3}$. The results of this computation are stored in a temporary table, along with other fields from Table3 that are referenced in SEQ. This operation results in:

```
forelem (k; k ∈ pTable3.field1)
  forelem (l; l ∈ pTable4.field1[Table3[k].field1])
    if (Table4[l].field2 == value4)
      $T = T \cup (\text{Table3}[k].field1)$
```

Note that all references in SEQ to Table3 must be rewritten to refer to $T$ instead.

### 6.3 Example

This section illustrates the usage of the transformations, by applying these on query 13 from the TPC-H benchmark\(^1\). The SQL code for query 13 is as follows:

```
select c_count,
  count(*) as custdist
from (  
  select c_custkey,
    count(o_orderkey)
  from customer left outer join orders on
    c_custkey = o_custkey
  and o_comment not like '%express%requests%'  
  group by
    c_custkey
) as c_orders (c_custkey, c_count)
  group by
    c_count
order by
```

\(^1\)Query 13 was chosen because of its size and usefulness to serve as an illustration. The other queries would have taken up too much space.
When this query is translated into the \textit{forelem} intermediate representation, the result is:

\begin{verbatim}
subquery0()
{
  forelem (i; i \in pCustomer)
  {
    forelem (j; j \in pOrders)
    {
      if (customer[i].c_custkey == orders[j].o_custkey &&
          !like(orders[j].o_comment, "%express%requests%")
        \mathcal{T} = \mathcal{T} \cup (customer[i].c_custkey, orders[j].o_orderkey)
      else
        \mathcal{T} = \mathcal{T} \cup (customer[i].c_custkey, nil)
    }
  }
  forelem (i; i \in p\mathcal{T})
  {
    \mathcal{G} = \mathcal{G} \cup (\mathcal{T}[i].c_custkey)
  }
  distinct(\mathcal{G})

  forelem (i; i \in p\mathcal{G})
  {
    count = 0;
    forelem (j; j \in p\mathcal{T}.c_custkey[\mathcal{G}[i].c_custkey])
    {
      if (\mathcal{T}[j].o_orderkey != nil)
        count++;
    }
    \mathcal{S} = \mathcal{S} \cup (\mathcal{G}[i].c_custkey, count)
  }

  return \mathcal{S};
}
\end{verbatim}

\begin{verbatim}
\mathcal{I} = \text{subquery0}();
forelem (i; i \in p\mathcal{I})
{
  \mathcal{R}_2 = \mathcal{R}_2 \cup (\mathcal{I}[i].c_count)
}
forelem (i; i \in p\mathcal{R}_2)
{
  \mathcal{R}_2 = \mathcal{R}_2 \cup (\mathcal{R}_2[i].c_count)
}
distinct(\mathcal{R}_2)

forelem (i; i \in p\mathcal{R}_2)
{
  count = 0;
  forelem (j; j \in p\mathcal{R}_2.c_count[\mathcal{R}_2[i].c_count])
  {
    count++;
  }
  \mathcal{R} = \mathcal{R} \cup (\mathcal{R}_2[i].c_count, count)
}
\end{verbatim}
As a first step, the Inline transformation is performed, which will inline the sub-query at the point where the subquery is called. For the above example this is trivial. Subsequently, the Loop Interchange transformation is considered. The only loop nest where Loop Interchange could possibly be applied is the loop nest over the Customer and Orders tables. Usually, Loop Interchange would be applied at this location such that the condition on o_comment can be tested in the outer loop. Note that in this case, the body of the if statement depends on both loop iterators and as such the statement cannot be moved to the other loop.

The next transformation that can be applied on this example is Table Propagation. In the first step, the loop creating table $\mathcal{T}$ is propagated to the consecutive loop nest accessing $\mathcal{T}$ and the loop creating $\mathcal{T}_2$ is propagated to the loops accessing $\mathcal{T}_2$.

```
forelem (i; i \in pCustomer) {
  forelem (j; j \in pOrders) {
    if (customer[i].c_custkey == orders[j].o_custkey &&
        !like(orders[j].o_comment, "%express%requests%")
      $\mathcal{T} = \mathcal{T} \cup (\text{customer}[i].c\text{.custkey}, \text{orders}[j].o\text{.orderkey})$
    else
      $\mathcal{T} = \mathcal{T} \cup (\text{customer}[i].c\text{.custkey}, \text{nil})$
  }
}
```

```
forelem (i; i \in pCustomer) {
  forelem (j; j \in pOrders) {
    if (customer[i].c_custkey == orders[j].o_custkey &&
        !like(orders[j].o_comment, "%express%requests%")
      $\mathcal{G} = \mathcal{G} \cup (\text{customer}[i].c\text{.custkey})$
    else
      $\mathcal{G} = \mathcal{G} \cup (\text{customer}[i].c\text{.custkey})$
  }
}
distinct(\mathcal{G})
```

```
forelem (i; i \in pG) {
  count = 0;
  forelem (j; j \in p\mathcal{T}.c\text{.custkey}[\mathcal{G}[i].c\text{.custkey}]) {
    if (\mathcal{T}[j].o\text{.orderkey} != \text{nil})
      count++;
  }
  $\mathcal{S} = \mathcal{S} \cup (\mathcal{G}[i].c\text{.custkey}, \text{count})$
}
```

```
forelem (i; i \in p\mathcal{S}) {
  $\mathcal{T}_2 = \mathcal{T}_2 \cup (\mathcal{G}[i].c\text{.count})$
}
```

```
forelem (i; i \in p\mathcal{S}) {
  $\mathcal{G}_2 = \mathcal{G}_2 \cup (\mathcal{G}[i].c\text{.count})$
}
distinct(\mathcal{G}_2)
```
6.3. Example

```java
forelem (i; i ∈ pG) {
    count = 0;
    forelem (j; j ∈ pG.c.count[%G[i].c.count]) {
        count++;
    }

    R = R ∪ (%G[i].c.count, count)
}

The transformation must be repeated several times for all propagations to be resolved. The result of the repeated application of the transformation is:

```code
forelem (i; i ∈ pCustomer) {
    forelem (j; j ∈ pOrders) {
        if (customer[i].c.custkey == orders[j].o.custkey &&
            !like(orders[j].o.comment, "%express%requests%")
            T = T ∪ (customer[i].c.custkey, orders[j].o.orderkey)
        else
            T = T ∪ (customer[i].c.custkey, nil)
    }
}
```
```
forelem (i; i ∈ pCustomer) {
    forelem (j; j ∈ pOrders) {
        if (customer[i].c.custkey == orders[j].o.custkey &&
            !like(orders[j].o.comment, "%express%requests%")
            G = G ∪ (customer[i].c.custkey)
        else
            G = G ∪ (customer[i].c.custkey)
    }
}
distinct(T)
```
```
forelem (i; i ∈ pS) {
    count = 0;
    forelem (ii; ii ∈ pCustomer.c.custkey[%S[i].c.custkey]) {
        forelem (jj; jj ∈ pOrders) {
            if (customer[ii].c.custkey == orders[jj].o.custkey &&
                !like(orders[jj].o.comment, "%express%requests%")
                row = (customer[ii].c.custkey, orders[jj].o.orderkey)
            else
                row = (customer[ii].c.custkey, nil)
            if (row.o.orderkey != nil)
                count++;
        }

        S = S ∪ (S[i].c.custkey, count)
    }
}
```
```
forelem (i; i ∈ pS) {

count = 0;
```
\[ T_2 = T_2 \cup (\mathcal{I}[i].c_{\text{count}}) \]

forelem (i; i ∈ p\(\mathcal{I}\)) {
    \( T_2 = T_2 \cup (\mathcal{I}[i].c_{\text{count}}) \)
    distinct(\( T_2 \))
}

forelem (i; i ∈ p\(\mathcal{I}\).distinct(c_{\text{count}})) {
    count = 0;
    forelem (j; j ∈ p\(\mathcal{I}\).c_{\text{count}}[\mathcal{I}[i].c_{\text{count}}]) {
        count++;
    }
    \( R = R \cup (\mathcal{I}[i].c_{\text{count}}, \text{count}) \)
}

Dead Code Elimination will remove statements that produce statements of which the results are not used:

forelem (i; i ∈ p\(\mathcal{I}\)) {
    \( \mathcal{G} = \mathcal{G} \cup (\text{customer}[i].\text{c_custkey}) \)
    distinct(\( \mathcal{G} \))
}

forelem (i; i ∈ p\(\mathcal{I}\)) {
    count = 0;
    forelem (ii; ii ∈ p\(\mathcal{I}\).c_{\text{custkey}}[\mathcal{G}[i].c_{\text{custkey}}]) {
        forelem (jj; jj ∈ p\(\mathcal{I}\).orders) {
            if (\text{customer}[ii].c_{\text{custkey}} == \text{orders}[jj].\text{o_custkey} &&
            !\text{like}(\text{orders}[jj].\text{o_comment}, "%express%requests%"))
                count++;
        }
    }
    \( \mathcal{I} = \mathcal{I} \cup (\mathcal{G}[i].\text{c_custkey}, \text{count}) \)
}

forelem (i; i ∈ p\(\mathcal{I}\).distinct(c_{\text{count}})) {
    count = 0;
    forelem (j; j ∈ p\(\mathcal{I}\).c_{\text{count}}[\mathcal{I}[i].c_{\text{count}}]) {
        count++;
    }
    \( R = R \cup (\mathcal{I}[i].c_{\text{count}}, \text{count}) \)
}

Note that in case all transformations are repeated, Table Propagation will propagate the loop iterating Customer and generating a table \( \mathcal{I} \), to the consecutive loop accessing \( \mathcal{I} \).
6.4 Optimization and Code Generation Strategies

In order to successfully optimize forelem loop nests using the transformations described in Section 6.2, a strategy is needed that determines in which order the transformations on the forelem loop nests are to be performed. The forelem framework uses the following strategy:

- First, subqueries are inlined, so that these can be considered in combination with the calling context.

- As a second step loops are reordered such that as many conditions as possible are tested in the outermost loops. Priority is given to move conditions that test against a constant value to the outermost loop. This step is a combination of the application of Loop Interchange with Loop Invariant Code Motion.

- Thirdly, opportunities for the application of Iteration Space Expansion are looked for. An example of such an opportunity is a loop iterating an index set with a condition on a field, of which the body computes an aggregate function. Iteration Space Expansion is followed by Loop Invariant Code Motion, because the loop computing the aggregate function is often made loop invariant by the Iteration Space Expansion transformation. Iteration Space Expansion is not applied on loops iterating temporary tables.

- The fourth step is to apply Table Propagation to prepare for the elimination of unnecessary temporary tables.

- Fifth, Index Extraction is performed on inner loops that iterate tables that could be removed from the loop nest.

- Finally, Dead Code Elimination is performed to remove any loop that computes unused results.

Experiments have been conducted with the queries from the TPC-H benchmark [91]. The different transformations that have been applied to each TPC-H query during the forelem optimization phase are shown in Table 6.1.

Another optimization strategy is to perform a brute-force exploration of the entire optimization space. This is useful, for example, for queries that are run many times on changing data so that the costly optimization effort is worth it. We plan to study brute-force exploration of the optimization search space in future work.

Code generation

Next to strategies for the application of transformations on the forelem intermediate representation, there are also strategies for the generation of efficient code from the forelem intermediate representation. These strategies are for a large part concerned with the selection of forelem loops for which index sets should be generated at run-time and the selection of efficient data structures for such index sets. The following rules are used for the code generation of index sets:
<table>
<thead>
<tr>
<th>Query #</th>
<th>Applied Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Table Propagation, LICM, Dead Code Elimination</td>
</tr>
<tr>
<td>2</td>
<td>Inline, Loop Interchange, LICM, Iteration Space Expansion, LICM, Index Extraction</td>
</tr>
<tr>
<td>3</td>
<td>Loop Interchange, LICM</td>
</tr>
<tr>
<td>4</td>
<td>Inline, Index Extraction</td>
</tr>
<tr>
<td>5</td>
<td>Loop Interchange, LICM, Index Extraction</td>
</tr>
<tr>
<td>6</td>
<td>None</td>
</tr>
<tr>
<td>7</td>
<td>Loop Interchange, LICM, Index Extraction</td>
</tr>
<tr>
<td>8</td>
<td>Loop Interchange, LICM, Index Extraction</td>
</tr>
<tr>
<td>9</td>
<td>Loop Interchange, LICM, Index Extraction, Table Propagation, Dead Code Elimination</td>
</tr>
<tr>
<td>10</td>
<td>Loop Interchange, Table Propagation</td>
</tr>
<tr>
<td>11</td>
<td>Inline, Loop Interchange, LICM, Table Propagation, Dead Code Elimination</td>
</tr>
<tr>
<td>12</td>
<td>Loop Interchange, LICM</td>
</tr>
<tr>
<td>13</td>
<td>Inline, Table Propagation, Dead Code Elimination</td>
</tr>
<tr>
<td>14</td>
<td>None</td>
</tr>
<tr>
<td>15</td>
<td>Inline, Loop Interchange, LICM, Table Propagation, Dead Code Elimination</td>
</tr>
<tr>
<td>16</td>
<td>Inline, Loop Interchange, LICM, Table Propagation, Dead Code Elimination</td>
</tr>
<tr>
<td>17</td>
<td>Iteration Space Expansion, LICM</td>
</tr>
<tr>
<td>18</td>
<td>Loop Interchange, LICM, Table Propagation, Dead Code Elimination</td>
</tr>
<tr>
<td>19</td>
<td>None</td>
</tr>
<tr>
<td>20</td>
<td>Inline, Iteration Space Expansion, LICM</td>
</tr>
<tr>
<td>21</td>
<td>Loop Interchange, LICM</td>
</tr>
<tr>
<td>22</td>
<td>Inline, LICM</td>
</tr>
</tbody>
</table>

Table 6.1: An overview of the transformations applied to each TPC-H query, in the order of application. The abbreviation LICM stands for Loop Invariant Code Motion.
1. Index sets without conditions address the fully array. No index set is generated in this case, instead the full array is iterated with subscripts $i \in [0, len)$.

2. Index sets that are used in multiple loop nests get priority in being generated.

3. The index set of the outer loop is never explicitly generated, as the outer loop is only iterated once.

4. For very small tables, index sets are not generated.

Different data structures are used as index set, such as flat arrays, hash tables or tree structures, depending on the properties of the index set. For example, if it is known that the field, for which an index set is created, has a unique value for each row in the array, a one-to-one-mapping is set up using a flat array or hash table. This property can be known to the code generator because the field was specified as primary key in the table schema, or the generated code detects at run-time that the table data satisfies this condition. For index sets that yield multiple subscripts balanced tree is used.

Additionally, the code generator can easily generate both row-wise and column-wise data access code. Within the forelem framework, a change from row-wise to column-wise layout is a trivial transformation. Which layout should be used is determined by the amount of fields in an array that are accessed.

### 6.5 Experimental Results

Experiments have been conducted using the queries from the TPC-H benchmark [91]. All queries were parsed into the forelem intermediate representation, optimized using the transformations described in this chapter and C/C++ code has been generated from the optimized AST. These executables access the database data through memory-mapped I/O. The execution time of the queries is compared to the execution time of the same queries as executed by PostgreSQL [80] and MonetDB [69].

All experiments have been carried out on an Intel Core 2 Quad CPU (Q9450) clocked at 2.66 GHz with 4 GB of RAM. The software installation consists out of Ubuntu 10.04.3 LTS (64-bit), which comes with PostgreSQL 8.4.9. The version of MonetDB used is 11.11.11 (Jul2012-SP2), which is the latest version that could be obtained from the MonetDB website [69] for use with this operating system.

On a TPC-H data set of scale factor 1.0, all queries were run with PostgreSQL, forelem-generated code and MonetDB. The execution times of the different queries in milliseconds are shown in Figure 6.1. PostgreSQL queries that took longer than 30 seconds to complete have been omitted from the figure for clarity. In the majority of cases, the forelem-optimized implementations have an execution time in the same order of magnitude as MonetDB, in a few cases even surpassing it.

MonetDB and the forelem-generated code, have also been tested on a dataset with scale factor 10.0. The execution times of the different queries in seconds are shown in Figure 6.2. In more than half of the queries, the forelem-optimized code performs the query with performance comparable to or faster than MonetDB.
Figure 6.1: Execution time in milliseconds of the TPC-H queries performed against a scale factor 1.0 dataset. The PostgreSQL results for Q1, Q2, Q17, Q20, Q21 were all beyond 31 seconds and were omitted from the figure for clarity.
Figure 6.2: Execution time in seconds of the TPC-H queries performed against a scale factor 10.0 dataset.
MonetDB is clearly faster in a third of the queries. We intend to address this gap in future work, by improving the used optimization strategies.

6.6 Conclusions

In this chapter, the optimization of database queries using compiler transformations has been described. This optimization process is carried out in the *forelem* framework. The *forelem* framework provides an intermediate representation in which queries can be naturally expressed and on which compiler transformations can be applied to optimize the loop nest. Compiler transformations that are currently implemented within the *forelem* framework were illustrated and strategies for the application of these transformations were discussed.

Experimentation using the queries from the TPC-H benchmark shows that the queries that were optimized using compiler transformations within the *forelem* framework are capable of achieving similar performance to that of contemporary database systems. However, while the *forelem* framework has been designed to provide full integral optimization, the *forelem* framework is still able to reach performance comparable to contemporary database systems.