A Versatile Tuple-Based Optimization Framework

Rietveld, K.F.D.

CHAPTER 5

Forelem Extensions for Aggregate Queries

This chapter proposes a method to express aggregation queries as *forelem* loop nests. An aggregation query is characterized by function calls into different stages of the aggregate function. These stages are defined, such that they can be used from an *forelem* loop nest to implement an aggregation query. Subsequently, before we can discuss group-by queries which depend heavily on aggregation, we introduce a syntax for working with the *distinct* keyword found in SQL. Typically, duplicate elimination is performed as the last operation during query evaluation. We propose that under several conditions, the *distinct* operation might be moved into the index sets eliminating the separate loop for duplicate elimination. Finally, we introduce a strategy for expressing group-by queries. We show that there are many opportunities to apply the transformations proposed in Chapters 3 and 4. In some cases it is possible to reduce the group-by query to a single *forelem* loop nest.

5.1 Expressing Aggregate Functions

An aggregate function typically has three stages: initialization, update and finalization. The stages serve to initialize any variables, update the variables for each tuple that is processed and to come to a final result. Not all aggregate functions have to implement all three stages. For example, to implement the `COUNT` aggregate, it is sufficient to implement initialization (to set the accumulator variable to zero) and update. To implement `AVG` it is also necessary to implement the finalization stage to perform the division of the sum.

For use within *forelem* loop nests we supply the following functions which represent the stages of an aggregate function:

- `agg_init (handle, agg_func)` initializes the given handle with the given aggregation function.

- `agg_step (handle, agg_func, value)` performs the step stage on the given handle, with the provided aggregate function and value derived from the current tuple.
- `agg_finish (handle, agg_func)` finishes the aggregate computation.
- `agg_result (handle)` returns the computed aggregate value for the handle.

At a later stage in optimization, these functions are replaced with inline variants of the called aggregate function. For the `COUNT` aggregate this means that `agg_init` is replaced with an assignment of the value zero to a variable to initialize the computation, `agg_step` is replaced with a simple value increment and `agg_finish` is replaced with a no-op. By inlining the actual operations, the *forelem* loop nests can be further optimized.

Let us consider the query

```
SELECT AVG (S.age)
FROM Sailors S
```

which performs the *average* aggregate function. We write this query as a *forelem* loop nest as follows using the 4 functions representing the aggregate stages:

```
agg_init(agg1, avg);
forelem (i; i ∈ pS)
  agg_step(agg1, avg, S[i].age);
agg_finish(agg1, avg);
ℛ = ℛ ∪ (agg_result(agg1))
```

When we append a `WHERE` clause to this query, for example `WHERE S.rating = 10`, it is sufficient to replace the use of the index set `pS` in the *forelem* loop with `pS.rating[10]`. Inlining the code performing the different stages of the aggregate function, we obtain:

```
agg1.sum = 0;
agg1.count = 0;
forelem (i; i ∈ pS)
{
  agg1.sum += S[i].age;
  agg1.count++;
}
agg1.result = agg1.sum / agg1.count;
ℛ = ℛ ∪ (agg1.result)
```

From this code sample it is clear that the loop body computes the sum of the vector consisting out of all `age` fields in `S` (the entire table `S` is iterated by index set `pS`). Similarly, the length of this vector is determined by incrementing the `count` variable in the loop body. We described in Section 1.1 that vectorizing compilers will recognize this pattern as reduction operator. The loop thus presents a vectorization opportunity for the optimizing compiler after the *forelem* code has been translated to C code. Without inlining, this opportunity would not have appeared.

5.2 Specification of distinct

Before we can describe how group-by queries are expressed as a *forelem* loop nest, we have to introduce syntax for handling `DISTINCT`. When the `DISTINCT` keyword, referred to as a set quantifier, is specified, redundant duplicate rows will be eliminated from the result table [45]. The keyword always operates on full tuples and it is not possible to perform distinct on a single specified column.

Given a temporary table $\mathscr{T}$, p$\mathscr{T}$.`distinct` specifies the index set on $\mathscr{T}$ that contains unique rows. Duplicates are not present in this index set. Corresponding to the SQL standard [45], the duplicate elimination is performed on the result table. Let us consider the query:

```
SELECT DISTINCT S.sname, S.age
FROM Sailors S
```

This results in:

```
forelem (i; i ∈ pS)
  𝒯 = 𝒯 ∪ (S[i].sname, S[i].age)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ 𝒯[i]
```

In certain cases, it is possible to eliminate the loop iterating over the unique rows of the result set. For this particular example, the loop over `S` does not have any conditions on `pS`. This makes it possible to perform the *distinct* operation when iterating over `pS`. Important is that this operation is applied on just the *sname* and *age* fields which are subsequently projected into the result table, instead of on the full tuples. If the operation is performed on the full tuples, tuples with equal *sname* and *age* but different values for the other fields of the table will still be duplicated in the result table.

The *distinct* syntax assumes by default that the *distinct* operation should be applied to all fields of the table. To limit the operation of the *distinct* keyword to specific fields, one can suffix a tuple of field names to the specification of *distinct* in the index set.

For our example this means that we suffix the *distinct* keyword with the fields *sname* and *age*. This results in the following condensed representation of the same query:

```
forelem (i; i ∈ pS.distinct(sname,age))
  ℛ = ℛ ∪ (S[i].sname, S[i].age)
```

We observe that by applying this transformation, the second loop has been eliminated. Naturally, the reverse transformation is also possible. By moving *distinct* back into a separate loop, application of other transformations is enabled that would otherwise be prevented due to the presence of *distinct* in the index set.

Note that elimination of the second loop for performing duplicate elimination is not always advantageous. Duplicate elimination is an expensive operation that is preferably applied on a table which is as small as possible. In certain cases, moving *distinct* into an index set is beneficial, specially when the operation is moved

to be applied on a smaller table, or when *distinct* is contained in a pre-computed index set.

The correctness of this transformation can be verified using relational algebra[1]. The original loop nest, with an additional loop for eliminating the duplicates, is expressed as:

$$\delta(\pi_{sname,age}(S))$$

In terms of relational algebra, we will express the *distinct* keyword suffixed with specific fields as a projection operation on these specific fields followed by duplicate elimination. The transformed loop nest is then expressed as:

$$\delta(\pi_{sname,age}(S))$$

which equals the expression for the original loop nest.

If in a single-level *forelem* loop all conditions are contained in the index set, it is possible to move the *distinct* operation to the index set. The *distinct* operation must then be limited to fields that will be added, or projected, to the result table. It is clear that this is an extension of the transformation on a loop nest without conditions.

As an example, consider:

```
forelem (i; i ∈ pS.age[18])
  𝒯 = 𝒯 ∪ (S[i].sname)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ (𝒯[i].sname)
```

with the following corresponding relational algebra expression:

$$\pi_{sname}(\delta(\pi_{sname}(\sigma_{age=18}(S))))$$

which can be simplified to:

$$\delta(\pi_{sname}(\sigma_{age=18}(S)))$$

The loop nest can be transformed into the following loop nest:

```
forelem (i; i ∈ pS.distinct(sname).age[18])
  𝒯 = 𝒯 ∪ (S[i].sname)
```

Note that in this syntax the *distinct* operation is performed after the selection, so the loop performs the following expression:

$$\pi_{sname}(\delta(\pi_{sname}(\sigma_{age=18}(S))))$$

where we can eliminate the outer projection again:

$$\delta(\pi_{sname}(\sigma_{age=18}(S)))$$

As a result, this loop is equal to the initial loop nest consisting of two loops.

[1]We use the extended relational algebra proposed by Dayal et. al. [29] which defines relations and operations on these relations in terms of multisets instead of sets. Furthermore, an explicit operator is introduced for elimination duplicates: δ.

If the loop body to which the *distinct* operation is moved contains an *if*-statement, the conditions under which this transformation can be carried out are limited. This makes sense, because the *if*-statement is now performed after the duplicate elimination has been done, contrary to the above example. The *if* statement must resemble a selection operation and when the fields used in the selection do not end up in the result table, the selection test must use the equality operator. The *distinct* operation must be applied to the fields that are added to the result tuple **and** the fields used in the comparison.

To illustrate this, consider the following example corresponding to $\delta(\pi_{sname}(\sigma_{age=18}(S)))$:

```
forelem (i; i ∈ pS)
  if (S[i].age == 18)
    𝒯 = 𝒯 ∪ (S[i].sname)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ (𝒯[i].sname)
```

Transformed into:

```
forelem (i; i ∈ pS.distinct(sname,age))
  if (S[i].age == 18)
    ℛ = ℛ ∪ (S[i].sname)
```

Consider as intermediate step a loop nest which has *distinct* as part of the index set and a separate loop for performing duplicate elimination. The relational algebra expression for such a loop nest is:

$$\delta(\pi_{sname}(\sigma_{age=18}(\delta(\pi_{sname,age}(S)))))$$

This equation can be obtained from the equation corresponding to the original loop by applying the properties of the algebra described in [29]. δ moves past π and δ commutes with σ. Secondly, we are free to remove columns that will not be projected or selected on further on.

To delete the *distinct* operator at the end of the chain (so at the left of the expression), either the projection at the end of the chain does not eliminate any new columns, which is essentially the case handled earlier in this section, or the projection does not introduce any new duplicates. The use of the equality operator during the selection in this case is crucial. The only way two tuples consisting of an *sname* and *age* field with the same values for *sname* can be distinct is to have different values for *age*. Since all tuples will have the value 18 for *age* after selection, all values for *sname* are distinct and the *age* column can be dropped without problems.

Clearly, this does not hold for other operators. Consider the use of the $<$ operator instead of equality. For a selection on $age < 18$ all tuples with $age < 18$ qualify, even if *sname* is equal. After this selection, tuples are present with equal *sname*.

After dropping the *age* column, the result is the following expression, which is indeed equal to the expression corresponding to the transformed loop nest:

$$\pi_{sname}(\sigma_{age=18}(\delta(\pi_{sname,age}(S))))$$

To double-level loop nests similar transformations can be applied Let us consider the following loop nest:

```
forelem (i; i ∈ pB.color["red"])
  forelem (j; j ∈ pR.bid[B[i].bid])
    𝒯 = 𝒯 ∪ (R[j].bid)
forelem (j; j ∈ p𝒯.distinct)
  ℛ = ℛ ∪ (𝒯[i].bid)
```

With the following corresponding relational algebra expression:

$$\pi_{R.bid}(\delta(\pi_{R.bid}(R \bowtie_{B.bid=R.bid} (\sigma_{B.color="red"}(B)))))$$

We apply properties from [29] to move δ past π and to distribute δ over $\bowtie$:

$$\delta(\pi_{R.bid}(\delta(R) \bowtie_{B.bid=R.bid} \delta(\sigma_{B.color="red"}(B))))$$

To be able to remove the distinct elimination at the end of the chain, we must ensure that the result of the join contains distinct values of *bid* because the final projection is only on *bid*. This is possible when both R and $\sigma_{B.color}(B)$ contain distinct values of *bid* before the join. Because no other fields are needed for the execution of this loop nest, we move the projection inside the distinct eliminations that take place before the join:

$$\pi_{R.bid}(\delta(\pi_{R.bid}(R)) \bowtie_{B.bid=R.bid} \delta(\pi_{B.bid}(\sigma_{B.color="red"}(B)))))$$

This corresponds with the following loop nest:

```
forelem (i; i ∈ pB.distinct(bid).color["red"])
  forelem (j; j ∈ pR.distinct(bid).bid[B[i].bid])
    ℛ = ℛ ∪ (R[j].bid)
```

To describe a case where *distinct* cannot be moved to the index set, we consider the query:

```
SELECT DISTINCT R.date
FROM Reserves R
WHERE R.bid = B.bid AND B.color = "red"
```

written in *forelem* as:

```
forelem (i; i ∈ pB.color["red"]))
  forelem (j; j ∈ pR.bid[B[i].bid])
    𝒯 = 𝒯 ∪ (R[j].date)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ 𝒯[i]
```

Let us look at the corresponding relational algebra expression, where the δ operator has already been distributed over the join:

$$\delta(\pi_{R.date}(\delta(R) \bowtie_{B.bid=R.bid} \delta(\sigma_{B.color="red"}(B))))$$

In order to eliminate δ at the end of the chain, we must project on *B.bid* and *R.bid*,*R.date* before the join. Only the date is projected into the result relation. This final operation will introduce duplicates: consider reservations for a different boat (*bid*) at the same date. So, in this case, we cannot eliminate the second loop performing duplicate elimination.

We can, however, eliminate the separate duplicate elimination loop after first performing a different transformation. When the Loop Collapse transformation described in Section 3.3.5 is performed, the result is:

```
forelem (i; i ∈ pB×R.(color^B,bid^R).[("red", bid^B)]))
  𝒯 = 𝒯 ∪ (B×R[i].date^R)
forelem (i; i ∈ p𝒯.distinct)
  ℛ = ℛ ∪ 𝒯[i]
```

Now the separate loop for *distinct* can be eliminated by moving the operation to the index set, because we can apply all conditions prior to the duplicate elimination. This also works if the conditions are specified in an *if*-statement instead and we apply *distinct* on all fields used.

5.3 Group-by queries

A group-by query groups tuples of a table by one or more fields, referred to as grouping columns. The values of other columns in the tuples can be aggregated using aggregate functions. Different methods for performing a group-by exist and an appropriate one is usually selected by the query optimizer depending on how table data is to be processed. These methods include performing the grouping operation by hashing and sorting an intermediate table followed by discovering and aggregating the groups.

We do not want to tie ourselves to a particular evaluation strategy for group-by queries, so the exact iteration patterns remain encapsulated in the *forelem* loops. Therefore our aim is to write a group-by query solely using *forelem* loops. In essence, a group-by query iterates over all groups identified by the grouping columns. Three stages are distinguished:

1. A temporary table $\mathscr{T}$ is created containing the selected columns of tuples adhering to an optionally specified `WHERE` clause.
2. The groups are extracted from this temporary table based on the specified grouping columns and stored in $\mathscr{G}$.
3. For each group in turn, we iterate over the group's members stored in $\mathscr{T}$ and perform the requested aggregate functions.

The three stages are written as three *forelem* loop nests. Subsequently, transformations can be applied, such as those described in Chapters 3 and 4. A potential result is that all three loops are merged into a single loop nest.

As an example, let us consider the query:

```
SELECT S.rating, MIN(S.age)
FROM Sailors S
GROUP BY S.rating
```

which we first express as three *forelem* loops:

```
forelem (i; i ∈ pS)
  𝒯 = 𝒯 ∪ (S[i].rating, S[i].age)

forelem (i; i ∈ p𝒯)
  𝒯2 = 𝒯2 ∪ (𝒯[i].rating)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, min)
  forelem (j; j ∈ p𝒯.rating[𝒢[i].rating])
    agg_step(agg1, min, 𝒯[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (𝒢[i].rating, agg_result(agg1))
}
```

We then apply a number of transformations on these loops to attempt to merge them into a single loop nest. In particular we will apply Temporary Table Reduction as described in Section 4.4.2. First, the first loop is duplicated such that the second and third loop nests, each using the results generated by the first loop, get a copy:

```
forelem (i; i ∈ pS)
  𝒯 = 𝒯 ∪ (S[i].rating, S[i].age)
forelem (i; i ∈ p𝒯)
  𝒯2 = 𝒯2 ∪ (𝒯[i].rating)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, min)
  forelem (j; j ∈ pS)
    𝒯3 = 𝒯3 ∪ (S[j].rating, S[j].age)
  forelem (j; j ∈ p𝒯3.rating[𝒢[i].rating])
    agg_step(agg1, min, 𝒯3[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (𝒢[i].rating, agg_result(agg1))
}
```

Now, we can apply Temporary Table Reduction to eliminate the generation of $\mathscr{T}$ and $\mathscr{T}_3$. For the third loop nest, this is accomplished by moving the index set conditions to *if*-statements, performing the reduction and moving the *if*-statements back to index set conditions.

```
forelem (i; i ∈ pS)
  𝒯2 = 𝒯2 ∪ (S[i].rating)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, min)
  forelem (j; j ∈ pS.rating[𝒢[i].rating])
    agg_step(agg1, min, S[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (𝒢[i].rating, agg_result(agg1))
}
```

On the first loop nest, it is now possible to eliminate the separate duplicate elimination loop using techniques described in Section 5.2.

```
forelem (i; i ∈ pS.distinct(rating))
  𝒢 = 𝒢 ∪ (S[i].rating)

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, min)
  forelem (j; j ∈ pS.rating[𝒢[i].rating])
    agg_step(agg1, min, S[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (𝒢[i].rating, agg_result(agg1))
}
```

Finally, another Temporary Table Reduction can be performed to merge both loop nests into one, eliminating the generation of temporary table $\mathscr{G}$.

```
forelem (i; i ∈ pS.distinct(rating))
{
  agg_init(agg1, min)
  forelem (j; j ∈ pS.rating[S[i].rating])
    agg_step(agg1, min, S[j].age)
  agg_finish(agg1, min)
  ℛ = ℛ ∪ (S[i].rating, agg_result(agg1))
}
```

Next, let us consider a more complicated example involving two tables and a `WHERE` clause:

```
SELECT B.bid, COUNT(*) AS reservationcount
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = "red"
GROUP BY B.bid
```

As we have discussed, the WHERE clause will be performed by the first loop. We express this query using three *forelem* loops for the three stages as follows:

```
forelem (i; i ∈ pB.color["red"])
  forelem (j; j ∈ pR.bid[B[i].bid])
    𝒯 = 𝒯 ∪ (B[i].bid, R[j].*)

forelem (i; i ∈ p𝒯)
  𝒯2 = 𝒯2 ∪ (𝒯[i].B.bid)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, count)
  forelem (j; j ∈ p𝒯.B.bid[𝒢[i].B.bid])
    agg_step(agg1, count)
  agg_finish(agg1, count)
  ℛ = ℛ ∪ (𝒢[i].B.bid, agg_result(agg1))
}
```

Note that two fields named *bid* are added to $\mathscr{T}$, to avoid confusion the fields are named *B.bid* and *R.bid*.

We use the same approach as used with the previous example: first duplicate the loop nest generating $\mathscr{T}$, and second use Temporary Table Reduction to merge this in the two remaining loop nests.

```
forelem (i; i ∈ pB.color["red"])
  forelem (j; j ∈ pR.bid[B[i].bid])
    𝒯2 = 𝒯2 ∪ (B[i].bid)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, count)
  forelem (ii; ii ∈ pB.(bid,color)[(𝒢[i].bid,"red")])
    forelem (jj; jj ∈ pR.bid[B[ii].bid])
      agg_step(agg1, count)
  agg_finish(agg1, count)
  ℛ = ℛ ∪ (𝒢[i].bid, agg_result(agg1))
}
```

Using the technique discussed in Section 5.2 we can eliminate the separate loop performing distinct elimination that follows the first loop nest:

```
forelem (i; i ∈ pB.distinct(bid).color["red"])
  forelem (j; j ∈ pR.distinct(bid).bid[B[i].bid])
    𝒢 = 𝒢 ∪ (B[i].bid)

forelem (i; i ∈ p𝒢)
{
  agg_init(agg1, count)
  forelem (ii; ii ∈ pB.(bid,color)[(𝒢[i].bid,"red")])
    forelem (jj; jj ∈ pR.bid[B[ii].bid])
      agg_step(agg1, count)
  agg_finish(agg1, count)
  ℛ = ℛ ∪ (𝒢[i].bid, agg_result(agg1))
}
```

Finally, we can eliminate the temporary table $\mathscr{G}$:

```
forelem (i; i ∈ pB.distinct(bid).color["red"])
  forelem (j; j ∈ pR.distinct(bid).bid[B[i].bid])
  {
    agg_init(agg1, count)
    forelem (ii; ii ∈ pB.(bid,color)[(B[i].bid,"red")])
      forelem (jj; jj ∈ pR.bid[B[ii].bid])
        agg_step(agg1, count)
    agg_finish(agg1, count)
    ℛ = ℛ ∪ (B[i].bid, agg_result(agg1))
  }
```

5.4 Having keyword

With the *having* keyword a condition can be specified that will be tested against each group. The condition usually only references grouping columns. This condition can only be tested after all members of a group have been processed. The condition cannot be moved into the index set of the enclosing loop.

As an example, we can extend the query used in the previous section to include a `HAVING` clause, specifying that only boats with more than 5 reservations should appear in the result table:

```
SELECT B.bid, COUNT(*) AS reservationcount
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = "red"
GROUP BY B.bid
HAVING COUNT(*) > 5
```

Because a `COUNT` aggregate is already performed, we do not have to introduce an additional aggregate computation. Before the tuple is added to the result set, we add a test for the *having* condition:

```
forelem (i; i ∈ pB.distinct(bid).color["red"])
  forelem (j; j ∈ pR.distinct(bid).bid[B[i].bid])
  {
    agg_init(agg1, count)
    forelem (ii; ii ∈ pB.(bid,color)[(B[i].bid,"red")])
      forelem (jj; jj ∈ pR.bid[B[ii].bid])
        agg_step(agg1, count)
    agg_finish(agg1, count)
    if (agg_result(agg1) > 5)
      ℛ = ℛ ∪ (B[i].bid, agg_result(agg1))
  }
```

After this addition, the *forelem* loop nest now computes the desired result.

5.5 Example

In this section we demonstrate how the techniques discussed in this chapter can be applied to a real-world code example. The following code fragment is based on the file *AboutMe.php* from the RUBiS [75] benchmark. The code fragment is written in pseudocode similar to PHP and edited for clarity.

```
$bidsResult =
    mysql_query("SELECT item_id, bids.max_bid FROM bids, items
                 WHERE bids.user_id=$userId AND bids.item_id=items.id
                 AND items.end_date > NOW()
                 GROUP BY item_id");
if (mysql_num_rows($bidsResult) == 0)
    print("<h2>You did not bid on any item.</h2>\n");
else
{
    print("<h3>Items you have bid on.</h3>\n");

    while ($bidsRow = mysql_fetch_array($bidsResult))
    {
        $maxBid = $bidsRow["max_bid"];
        $itemId = $bidsRow["item_id"];
        $itemResult =
            mysql_query("SELECT * FROM items WHERE id=$itemId");

        $currentPriceResult =
            mysql_query("SELECT MAX(bid) AS bid FROM bids ".
                        "WHERE item_id=$itemId");
        $currentPriceRow = mysql_fetch_array($currentPriceResult);
        $currentPrice = $currentPriceRow["bid"];
        if ($currentPrice == null)
            $currentPrice = "none";
```

```
        $itemRow = mysql_fetch_array($itemResult);

        $itemName = $itemRow["name"];
        $itemInitialPrice = $itemRow["initial_price"];
        $quantity = $itemRow["quantity"];
        $itemReservePrice = $itemRow["reserve_price"];
        $startDate = $itemRow["start_date"];
        $endDate = $itemRow["end_date"];
        $sellerId = $itemRow["seller"];

        $sellerResult =
            mysql_query("SELECT nickname FROM users " .
                        "WHERE id=$sellerId")

        $sellerRow = mysql_fetch_array($sellerResult);
        $sellerNickname = $sellerRow["nickname"];

        print("<TR><TD>" .
              "<a href=\"/PHP/ViewItem.php?itemId="
              .$itemId."\">".$itemName.
              "<TD>".$itemInitialPrice."<TD>".$currentPrice."<TD>"
              .$maxBid."<TD>".$quantity.
              "<TD>".$startDate."<TD>".$endDate.
              "<TD><a href=\"/PHP/ViewUserInfo.php?" .
              "userId=".$sellerId."\">".$sellerNickname.
              "</a>\n");

        mysql_free_result($sellerResult);
        mysql_free_result($currentPriceResult);
        mysql_free_result($itemResult);
    }
    mysql_free_result($bidsResult);
}
```

As a first step, all SQL queries that are performed by calling the DBMS API are replaced with *forelem* loop nests which execute in process. The code fragment contains four queries. We will rewrite these queries as *forelem* loop nests and perform preliminary transformations on these queries in turn. After that, we place the loop nests into the code fragment. The first query is:

```
SELECT item_id, bids.max_bid FROM bids, items
WHERE bids.user_id=$userId AND bids.item_id=items.id
AND items.end_date >= NOW()
GROUP BY item_id
```

This query is written as a *forelem* loop nest using the strategy discussed in Section 5.3:

```
forelem (i; i ∈ pBids.user_id[$userId])
  forelem (j; j ∈ pItems.(id,end_date)[(Bids[i].item_id,[NOW(),∞)])
    𝒯 = 𝒯 ∪ (Bids[i].item_id, Bids[i].max_bid)

forelem (i; i ∈ p𝒯)
  𝒯2 = 𝒯2 ∪ (𝒯[i].item_id)
forelem (i; i ∈ p𝒯2.distinct)
  𝒢 = 𝒢 ∪ 𝒯2[i]

forelem (i; i ∈ p𝒢)
{
  forelem (j; j ∈ p𝒯.item_id[𝒢[i].item_id])
    r = (𝒯[j].item_id, 𝒯[j].max_bid)
  ℛ = ℛ ∪ r
}
```

Note that $[NOW(), \infty)$ indicates the range in which the value of field *end_date* must lie. And with the described transformations, we can write the query as a single loop nest:

```
forelem (i; i ∈ pBids.distinct(item_id).user_id[$userId])
  forelem (j; j ∈ pItems.distinct(id).(id,end_date)
                              [(Bids[i].item_id,[NOW(),∞))])
  {
    forelem (ii; ii ∈ pBids.(user_id,item_id)
                                [($userId,Bids[i].item_id)])
      forelem (jj; jj ∈ pItems.(id,end_date)
                                  [(Bids[ii].item_id,[NOW(),∞))])
        r = (Bids[ii].item_id, Bids[ii].max_bid)
    ℛ = ℛ ∪ r
  }
```

The second query to be considered is:

```
SELECT * FROM items WHERE id=$itemId
```

which is written as:

```
forelem (i; i ∈ pItems.id[$itemId])
  ℛ = ℛ ∪ Items[i]
```

The third query contains an aggregate function:

```
SELECT MAX(bid) AS bid FROM bids WHERE item_id=$itemId
```

Using the technique described in Section 5.1 we can express the query using *forelem* loops as follows:

```
agg_init(agg1, max);
forelem (i; i ∈ pBids.item_id[$item_id])
  agg_step(agg1, max, Bids[i].bid);
agg_finish(agg1, max);
ℛ = ℛ ∪ (agg_result(agg1))
```

The aggregate operation can subsequently be inlined:

```
agg1.result = 0;
forelem (i; i ∈ pBids.item_id[$item_id])
  if (agg1.result == 0 || agg1.result < Bids[i].bid)
    agg1.result = Bids[i].bid;
ℛ = ℛ ∪ (agg1.result)
```

Finally, the fourth query:

```
SELECT nickname FROM users WHERE id=$sellerId
```

is easily converted to:

```
forelem (i; i ∈ pUsers.id[$sellerId])
  ℛ = ℛ ∪ (Users[i].nickname)
```

We now rewrite the code fragment with the *forelem* loops for the four queries:

```
forelem (i; i ∈ pBids.distinct(item_id).user_id[$userId])
  forelem (j; j ∈ pItems.distinct(id).(id,end_date)
                        [(Bids[i].item_id,[NOW(),∞))])
  {
    forelem (ii; ii ∈ pBids.(user_id,item_id)
                          [($userId,Bids[i].item_id)])
      forelem (jj; jj ∈ pItems.(id,end_date)
                            [(Bids[ii].item_id,[NOW(),∞))])
        r = (Bids[ii].item_id, Bids[ii].max_bid)
    ℛ1 = ℛ1 ∪ r
  }
if (is_empty (ℛ1))
    print("<h2>You did not bid on any item.</h2>\n");
else
{
    print("<h3>Items you have bid on.</h3>\n");

    while ($bidsRow ∈ ℛ1)
    {
        $maxBid = $bidsRow["max_bid"];
        $itemId = $bidsRow["item_id"];

        forelem (i; i ∈ pItems.id[$itemId])
          ℛ2 = ℛ2 ∪ Items[i];
```

```
        agg1.result = 0;
        forelem (i; i ∈ pBids.item_id[$item_id])
          if (agg1.result == 0 || agg1.result < Bids[i].bid)
            agg1.result = Bids[i].bid;
        ℛ3 = ℛ3 ∪ (agg1.result);

        $currentPriceRow = r ∈ ℛ3;
        $currentPrice = $currentPriceRow["bid"];
        if ($currentPrice == null)
            $currentPrice = "none";

        $itemRow = r ∈ ℛ2;

        $itemName = $itemRow["name"];
        $itemInitialPrice = $itemRow["initial_price"];
        $quantity = $itemRow["quantity"];
        $itemReservePrice = $itemRow["reserve_price"];
        $startDate = $itemRow["start_date"];
        $endDate = $itemRow["end_date"];
        $sellerId = $itemRow["seller"];

        forelem (i; i ∈ pUsers.id[$sellerId])
          ℛ4 = ℛ4 ∪ (Users[i].nickname)

        $sellerRow = r ∈ ℛ4;
        $sellerNickname = $sellerRow["nickname"];

        print("<TR><TD>" .
              "<a href=\"/PHP/ViewItem.php?itemId="
              .$itemId."\">".$itemName.
              "<TD>".$itemInitialPrice."<TD>".$currentPrice."<TD>"
              .$maxBid."<TD>".$quantity.
              "<TD>".$startDate."<TD>".$endDate.
              "<TD><a href=\"/PHP/ViewUserInfo.php?" .
              "userId=".$sellerId."\">".$sellerNickname.
              "</a>\n");
    }
}
```

We now apply Loop Merge to merge the *forelem* loop producing the tuples into result set $\mathscr{R}_1$ with the while loop consuming these tuples. Before this transformation can be applied, we must perform a preparatory transformation that moves the *if*-statement checking `is_empty` after the merged loop. The statements in the *else* clause before the while loop are moved into the loop body and made conditional. At the same time we perform an explicit table reduction which replaces references into the result set with direct references into the database table. Subsequently, Global Forward Substitution can be performed. This reduction is also

applied on the result set $\mathscr{R}_3$.

```
results = 0;
forelem (i; i ∈ pBids.distinct(item_id).user_id[$userId])
  forelem (j; j ∈ pItems.distinct(id).(id,end_date)
                          [(Bids[i].item_id,[NOW(),∞))])
  {
    forelem (ii; ii ∈ pBids.(user_id,item_id)
                              [($userId,Bids[i].item_id)])
      forelem (jj; jj ∈ pItems.(id,end_date)
                               [(Bids[ii].item_id,[NOW(),∞))])
        r = (Bids[ii].item_id, Bids[ii].max_bid)

    if (results == 0)
      print("<h3>Items you have bid on.</h3>\n");

    results++;

    forelem (iii; iii ∈ pItems.id[Bids[ii]["item_id"]])
      𝓡2 = 𝓡2 ∪ Items[iii]

    agg1.result = 0;
    forelem (iii; iii ∈ pBids.item_id[Bids[ii]["item_id"]])
      if (agg1.result == 0 || agg1.result < Bids[iii].bid)
        agg1.result = Bids[iii].bid;
    $currentPrice = agg1.result;

    if ($currentPrice == null)
        $currentPrice = "none";

    $itemRow = r ∈ 𝓡2;

    $itemName = $itemRow["name"];
    $itemInitialPrice = $itemRow["initial_price"];
    $quantity = $itemRow["quantity"];
    $itemReservePrice = $itemRow["reserve_price"];
    $startDate = $itemRow["start_date"];
    $endDate = $itemRow["end_date"];
    $sellerId = $itemRow["seller"];

    forelem (iii; iii ∈ pUsers.id[$sellerId])
      𝓡4 = 𝓡4 ∪ (Users[iii].nickname)

    $sellerRow = r ∈ 𝓡4;
    $sellerNickname = $sellerRow["nickname"];

    print("<TR><TD>" .
```

```
          "<a href=\"/PHP/ViewItem.php?itemId="
          .$itemId."\">".$itemName.
          "<TD>".$itemInitialPrice."<TD>".$currentPrice."<TD>"
          .$maxBid."<TD>".$quantity.
          "<TD>".$startDate."<TD>".$endDate.
          "<TD><a href=\"/PHP/ViewUserInfo.php?" .
          "userId=".$sellerId."\">".$sellerNickname.
          "</a>\n");
  }
}
if (results == 0)
  print("<h2>You did not bid on any item.</h2>\n");
```

Further optimizations are possible. For example, def-use analysis will detect that only a single row of $\mathscr{R}_2$ is used. The analysis will also detect that the condition `id == Bids[ii]["item_id"]` holds for all tuples iterated by iteration counter `jj`. Therefore, this *forelem* loop is unnecessary and the data can simply be obtained from `Items[jj]` instead.

Also from result set $\mathscr{R}_4$ a single tuple is used. Therefore, the loop generating this result set can be pruned to only iterate once. This can be accomplished either by using the *single* modifier described in Section 4.2 or by using an additional mask column as described in Section 3.3.5. After this transformation, explicit table reduction can be applied on this loop.

Finally, the fact that the tables *Bids* and *Items* are closely used together might indicate that the Loop Collapse transformation, described in Section 3.3.5 can be of use here. This will eliminate the two joins currently present in the loop nest and might open the road to further transformations. As an example, this has the potential to make it possible to eliminate the query computing the `MAX(bid)` aggregate.

5.6 Conclusions

In this chapter we demonstrated how aggregation queries can be written in terms of a *forelem* loop and introduced a strategy for expressing group-by queries as *forelem* loops. A syntax for duplicate elimination was introduced together with conditions under which the duplicate elimination can be moved to the *forelem* loops' index sets. We have demonstrated that the transformations introduced in the preceding chapters can be applied. Whereas a group-by query is first written as three *forelem* loop nests, it is in certain cases possible to transform this to a single *forelem* loop nest.

By means of an example, we have demonstrated that many potential optimizations exist that can take advantage of the described strategies and transformations. In the example, we were able to merge a code fragment containing a group-by query and three other queries into a single loop nest. Subsequently, the possibility was shown how one of the queries can be fully eliminated. There are further possibilities to optimize this loop nest for example by restructuring the tables using Loop Collapse.