Loop Elimination for Database Updates

Vadim Savenkov[1], Reinhard Pichler[1], and Christoph Koch[2]

[1] Vienna University of Technology
{savenkov,pichler}@dbai.tuwien.ac.at
[2] École Polytechnique Fédérale de Lausanne
christoph.koch@epfl.ch

Abstract. The additional expressive power of procedural extensions of query and update languages come at the expense of trading the efficient set-at-a-time processing of database engines for the much less efficient tuple-at-a-time processing of a procedural language. In this work, we consider the problem of rewriting for-loops with update statements into sequences of updates which do not use loops or cursors and which simultaneously carry out the action of several loop iterations in a set-at-a-time manner. We identify idempotence as the crucial condition for allowing such a rewriting. We formulate concrete rewrite rules for single updates in a loop and extend them to sequences of updates in a loop.

1 Introduction

To enhance the expressive power of SQL for querying and modifying data, the SQL standard proposes SQL/PSM as a Turing complete procedural extension of SQL. Most relational database management systems provide their own procedural programming language such as PL/pgSQL of PostgreSQL, PL/SQL of Oracle, SQL PL of IBM's DB2, Transact-SQL in Microsoft SQL Server, etc. The key feature of these extensions of SQL is to allow the definition of loops for iterating over relations with a cursor and to "parameterize" so to speak the action in the loop body by the current tuples of these relations. The additional expressive power however comes at the expense of trading the efficient set-at-a-time processing of SQL for the much less efficient tuple-at-a-time processing of a procedural language. For the sake of optimizing updates, the question naturally arises if a given sequence of updates necessarily has to be realized by a loop containing these updates or whether it would be possible to achieve the same effect with a sequence of simple updates that do not use loops or cursors.

In this paper we restrict ourselves to *for*-loops with updates. Our goal is to provide rewrite rules that allow one to transform for-loops with update statements into a sequence of simple updates that simultaneously carry out the action of several loop iterations in a set-at-a-time manner. To this end, we will first introduce an update language which we find convenient for our investigations and point out how update statements of this form can be represented in (standard) SQL. We then identify a crucial property of updates as a sufficient condition for the elimination of for-loops containing update statements, namely the *idempotence* of updates, i.e., applying the same update twice or more often yields the

G. Gottlob et al. (Eds.): BNCOD 2013, LNCS 7968, pp. 48–61, 2013.

same result as a single application of the update. Based on this condition, we shall define rewrite rules for unnesting a single update and also several successive updates in a for-loop. The elimination of nested loops with updates is thus also covered by successively applying the rewrite rules to each loop - starting with the innermost one.

Update optimization is an old topic in database research. A problem statement similar to ours was considered by Lieuwen and DeWitt in [5], who provided rules for optimizing for-loop statements in the database programming language O++. There, the authors focus on flattening the nested loops. In contrast, our approach allows for complete elimination of loops and replacing them with relational-style update commands. This problem has been also considered in [1] in the context of uncertain databases. The results in the present paper extend that work: in particular, we consider update commands in which arbitrary attributes can be referenced on the right-hand side of equalities in the *set*-clause, whereas in [1] only constants are supported.

Our transformation relies on the *idempotence* of update operations, which can be easily tested: the operation is idempotent if repeating it twice or more times has the same effect as applying it only once. The importance of the idempotence property for update optimization for the task of incremental maintenance of materialized views [4], has been identified in [3]. More recently, idempotent operations have been found useful also in a broader setting in the area of distributed systems [2,6]. Efficient and block-free methods of failure recovery are essential in distributed environments. The idempotence property ensures that such repeated evaluation is safe and does not change the semantics of a program.

Organization of the Paper and Summary of Results. In Section 2, we introduce a simple update language, which is convenient for our investigations, and establish its connection to (standard) SQL. In Section 3 we present our rewrite rule for eliminating a for-loop with a single update statement inside. Clearly, this rewrite rule is also applicable to nested loops starting with the innermost loop. The crucial condition for the applicability of our rewrite rule is the idempotence of the update. In Section 4, we formulate a sufficient condition for the elimination of for-loops with more than one update inside and present an appropriate extension of our rewrite rule.

2 Update Language

Suppose that we want to update some relation R whose schema $\mathbf{sch}(R)$ is given as $\mathbf{sch}(R) = \{A_1, \ldots, A_m\}$. In this paper, we restrict ourselves to updates which can be defined by a relation U with $\{A_1, \ldots, A_m, A'_1, \ldots, A'_m\} \subseteq \mathbf{sch}(U)$ in the following sense: the tuples affected by such an update are $T = \{r \mid \exists u \in U, \text{ s.t. } r.A_1 = u.A_1 \wedge \ldots \wedge r.A_m = u.A_m\}$, i.e., $T = R \ltimes U$. The new values to which the attributes $\bar{A} = (A_1, \ldots, A_m)$ of each tuple $r \in T$ are modified are defined by the components $\bar{A}' = (A'_1, \ldots, A'_m)$ of the corresponding tuple in U, i.e.: each $r \in T$ is modified to $\pi_{\bar{A}'}(\sigma_{U.\bar{A}=r}(U))$. Clearly, there must exist a functional dependency

$U.\bar{A} \to U.\bar{A}'$ to make sure that every tuple in T is mapped to precisely one value $U.\bar{A}'$. This leads to the definition of the following language of update statements:

Definition 1. *Let R and U be relations with $\boldsymbol{sch}(R) = \{A_1, \ldots, A_m\}$ and $\{A_1, \ldots, A_m, A'_1, \ldots, A'_m\} \subseteq \boldsymbol{sch}(U)$. Then the "update defined by U" is denoted as*

$$\boxed{\text{update R set } \bar{A} = U.\bar{A}' \text{ from } U \text{ where } R.\bar{A} = U.\bar{A};}$$

Such an update is called well-defined *if there exists a functional dependency $U.\bar{A} \to U.\bar{A}'$. In this case, the effect of this update is to replace each tuple r in $R \ltimes U$ by the uniquely determined value $U.\bar{A}'$, s.t. $r.\bar{A} = U.\bar{A}$.*

Note that the above definition imposes no restriction on the nature of the relation U. In particular, U may itself be defined by some query. In this case, the value of U immediately before updating R is considered as fixed throughout the update process. This is in line with the transactional semantics of SQL updates, i.e., changes made by an update are not visible to the update itself before the end of the update operation.

The proposed syntax is general enough to cover many practical cases. In particular, the updates of the form "update R set $\bar{A} = \bar{c}$ where ϕ", considered in [1], can be captured easily: Let Q_ϕ denote the semi-join query returning the tuples of R that have to be updated. In order to write the above update in the form: update R set $\bar{A} = U.\bar{A}'$ from U where $R.\bar{A} = U.\bar{A}$; we have to define the relation U. Suppose that $\mathbf{sch}(R) = \{A_1, \ldots, A_m, B_1, \ldots B_n\}$, which we abbreviate as $\bar{A}, \bar{B}$. Likewise, we write $\bar{A}', \bar{B}'$ to denote $\{A'_1, \ldots, A'_m, B'_1, \ldots B'_n\}$. Then $U(\bar{A}, \bar{B}, \bar{A}', \bar{B}')$ is defined by the following query (in logic programming notation):

$U(\bar{A}, \bar{B}, \bar{c}, \bar{B})$:- $R(\bar{A}, \bar{B}), Q_\phi$;

We give some further simple examples of updates below:

Example 1. Consider a relation R with attributes (A_1, A_2). An update operation that swaps the two attributes of R can be defined as

$$\text{update R set } \bar{A} = U.\bar{A}' \text{ from } U \text{ where } R.\bar{A} = U.\bar{A};$$

such that $U(A_1, A_2, A'_1, A'_2)$ is defined by the following query:

$U(A_1, A_2, A_2, A_1)$:- $R(A_1, A_2)$.

Now suppose that A_1, A_2 have numerical values. Moreover, suppose that we want to apply the above update only to tuples r in R where A_1 is an even number and $A_1 < A_2$ holds. Then we just have to modify the definition of relation U, namely:

$U(A_1, A_2, A_2, A_1)$:- $R(A_1, A_2)$, $A_1 < A_2$, $A_1 \bmod 2 == 0$.

More generally, suppose that R contains m attributes $(A_1, \ldots, A_m)$ and we want to swap the first two of them. Then $U(A_1, \ldots, A_m, A'_1, \ldots, A'_m)$ is defined as follows:

$U(A_1, A_2, A_3, \ldots, A_m, A_2, A_1, A_3, \ldots, A_m)$:- $R(A_1, \ldots, A_m)$. □

We conclude this section by describing a translation of updates formulated in our syntax to standard SQL. Consider an update of the form

$$\text{update R set } \bar{A} = U.\bar{A}' \text{ from } U \text{ where } R.\bar{A} = U.\bar{A};$$

where $\bar{A}$ denotes the attributes $\{A_1, \ldots, A_m\}$ of R. This update can be rewritten as follows:

update R set
A_1 = (select A'_1 from U where $R.\bar{A} = U.\bar{A}$)
...
A_m = (select A'_m from U where $R.\bar{A} = U.\bar{A}$)
where exists (select * from U where $R.\bar{A} = U.\bar{A}$).

where we write $R.\bar{A} = U.\bar{A}$ to abbreviate the condition $R.A_1 = U.A_1$ and ... and $R.A_m = U.A_m$. If the DBMS supports the extended update syntax (like 'update from' in PostgreSQL), then the SQL update statement becomes more concise:

update R set $A_1 = A'_1, \ldots, A_m = A'_m$
from U
where $R.\bar{A} = U.\bar{A}$.

Of course, in simple cases, the relation U does not have to be defined explicitly (e.g., as a view), as the following example illustrates:

Example 2. Consider relations R, S, P with $\mathbf{sch}(R) = \{A_1, A_2, A_3\}$, $\mathbf{sch}(S) = \{B_1, B_2\}$, and $\mathbf{sch}(P) = \{C_1, C_2\}$. Let an update be defined by the relation $U(A_1, A_2, A_3, A'_1, A'_2, A'_3)$, where U is defined as follows:

$$U(A_1, A_2, A_3, A_1, A'_2, A'_3) \text{ :- } S(A_2, A'_2), P(A_3, A'_3), A'_2 < A'_3.$$

Intuitively, S defines the update of the second component of R and P defines the update of the third component of R. Moreover, these updates may only be applied if the new value for the second component of R is less than for the third one. In SQL we get:

update R
set
  A_2 = (select $S.B_2$ as A'_2 from S, P
    where $S.B_1 = R.A_2$ and $P.C_1 = R.A_3$ and $S.B_2 < P.C_2$),
  A_3 = (select $P.C_2$ as A'_3 from S, P
    where $S.B_1 = R.A_2$ and $P.C_1 = R.A_3$ and $S.B_2 < P.C_2$)
where exists (select * from S, P
    where $S.B_1 = R.A_2$ and $P.C_1 = R.A_3$ and $S.B_2 < P.C_2$).

If the DBMS supports the extended update syntax, then the above update statement can be greatly simplified to:

update R
set $A_2 = S.B_2$, $A_3 = P.C_2$
from S, P
where $S.B_1 = A_2$ and $P.C_1 = A_3$ and $S.B_2 < P.C_2$ □

for ($\$t$ in Q){update R set $\bar{A} = U[\$t].\bar{A}'$ from $U[\$t]$ where $R.\bar{A} = U[\$t].\bar{A}$};
$\vdash$ update R set $\bar{A} = V.\bar{A}'$ from V where $R.\bar{A} = V.\bar{A}$;
s.t. $V = \bigcup_{t \in Q} U[t]$.

Fig. 1. Unnesting update programs

3 Loop Elimination

Recall from Definition 1 that, in this paper, we are considering updates defined by some relation U. Now suppose that an update occurs inside a loop which iterates over the tuples in some relation Q. Hence, in general, the update relation U depends on the current tuple t of Q. We thus write $U[t]$ to denote the value of U for a given tuple t of Q. In a loop over Q, the relation U is parameterized so to speak by the tuples in Q. We thus write $U[\$t]$ to denote the family of relations U that we get by instantiating the variable $\$t$ by the tuples t in Q. We thus have to deal with loops of the following form:

for ($\$t$ in Q){update R set $\bar{A} = U[\$t].\bar{A}'$ from $U[\$t]$ where $R.\bar{A} = U[\$t].\bar{A}$};

where $\mathbf{sch}(R) = \{A_1, \ldots, A_m\}$. Moreover, for every instantiation of $\$t$ to a tuple t over the schema $\mathbf{sch}(Q)$, $U[\$t]$ yields a relation whose schema contains the attributes $\{A_1, \ldots, A_m, A'_1, \ldots, A'_m\}$. The relation resulting from instantiating $\$t$ to t is denoted as $U[t]$. The semantics of the above loop is the following: *for each value $\$t$ in Q, peform the update of R using the update relation $U[\$t]$ according to Definition 1.*

Of course, updates may also occur inside nested loops. We thus get statements of the following form:

for ($\$t_1$ in Q_1) {
  for ($\$t_2$ in Q_2) {
    ...
      for ($\$t_n$ in Q_n) {update R
                              set $\bar{A} = U[\$t_1, \ldots, \$t_n].\bar{A}'$
                              from $U[\$t_1, \ldots, \$t_n]$
                              where $R.\bar{A} = U[\$t_1, \ldots, \$t_n].\bar{A}\}\ldots\}\}$;

such that, for every instantiation of $\$t_1, \ldots, \$t_n$ to tuples $t_1, \ldots, t_n$ over the schemas $\mathbf{sch}(Q_1), \ldots, \mathbf{sch}(Q_n)$, $U[\$t_1, \ldots, \$t_n]$ yields a relation whose schema contains the attributes $\{A_1, \ldots, A_m, A'_1, \ldots, A'_m\}$. The relation resulting from instantiating $\$t_1$ to t_1, ..., $\$t_n$ to t_n is denoted as $U[t_1, \ldots, t_n]$.

For unnesting updates, it suffices to provide a rule for transforming a single for-loop with update into an update statement without loop. In case of nested loops, this transformation has to be applied iteratively starting with the innermost for-loop. Such a rule can be found in Fig. 1. It is put into effect in the following example:

Example 3. Consider relations Department and Employee: $\mathbf{sch}(\mathsf{Department}) = \{\mathsf{dept\_id}, \mathsf{bonus}\}$, $\mathbf{sch}(\mathsf{Employee}) = \{\mathsf{empl\_id}, \mathsf{dept\_id}, \mathsf{base\_salary}, \mathsf{compensation}\}$. Using logic programming notation, we define the relations Q and U:

$$Q(Dept\_id, Bonus) \text{ :- } \mathsf{Department}(Dept\_id, Bonus)$$

$$U(Empl\_id, Dept\_id, Base\_sal, Comp,\\ Empl\_id, Dept\_id, Base\_sal, Comp')[\$t] \text{ :- } Comp' = Base\_sal \cdot (1 + \$t.\mathsf{bonus})$$

The following update loop increases the compensation of all employees:

for($\$t$ in Q)\
update Employee set compensation = $U[\$t]$.compensation′ from $U[\$t]$\
where $U[\$t]$.empl_id = Employee.empl_id and\
$U[\$t]$.dept_id = Employee.dept_id and\
$U[\$t]$.base_salary = Employee.base_salary and\
$U[\$t]$.compensation = Employee.compensation

For the sake of readability, in the *set*-clause of the update command we omit the assignments to the attributes which are not modified in $U[\$t]$. Applying the unnesting rule from Fig. 1, the update loop can be rewritten as the following command:

update Employee set compensation = V.compensation′ from V\
where V.empl_id = Employee.empl_id and\
V.dept_id = Employee.dept_id and\
V.base_salary = Employee.base_salary and\
V.compensation = Employee.compensation

Here, V is obtained from $U[\$t]$ by taking a join of $U[\$t]$ with Q and replacing $\$t$ in the body of U with Q.

$$V(Empl\_id, Dept\_id, Base\_sal, Comp, Empl\_id, Dept\_id, Base\_sal, Comp') \text{ :-}\\ Comp' = Base\_sal \cdot (1 + Q.Bonus),\\ Q(Dept\_id, Bonus)$$

It is easy to see that the above expression for V expresses exactly the one in Fig. 1, namely $V = \bigcup_{t \in Q} U[t]$. This expression can be further simplifed as

$$V(Empl\_id, Dept\_id, Base\_sal, Comp, Empl\_id, Dept\_id, Base\_sal, Comp') \text{ :-}\\ Comp' = Base\_sal \cdot (1 + \mathsf{Department}.Bonus),\\ \mathsf{Department}(Dept\_id, Bonus)$$ □

Remark 1. Note that to ensure that the update is well-defined, it might be necessary to inspect the particular instance of the join between the updated table and the update relation U. However, if certain integrity constraints are present in the schema, such inspection can be spared, since the desired key dependency may be inferred from the definition of the update relation U and existing schema constraints. For instance, for the update relation $U[\$t](\bar{A}, \bar{A}')$ in Example 3, the dependency $\bar{A} \to \bar{A}'$ has to be checked. Since the only modified

attribute of Employee is compensation, it suffices to check the functional dependency $\bar{A} \to \mathsf{compensation}'$, where $\bar{A}$ denotes the first four attributes of $U[\$t]$. Note that $\mathsf{compensation}'$ is determined by base_salary and Department.bonus. Moreover, note that empl_id and dept_id are respective primary keys in the Employee and Department tables. Then, also the functional dependency empl_id $\to$ bonus holds in the relation Employee $\bowtie$ Department. Therefore, the functional dependency empl_id $\to \mathsf{compensation}'$ holds in $U[\$t]$ and thus the respective update is well-defined irrespective of the database instance.

The following theorem gives a sufficient correctness criterion for the loop elimination from Fig. 1.

Theorem 1. *Let Q and R be relations with $\boldsymbol{sch}(Q) = \{B_1, \ldots, B_\ell\}$ and $\boldsymbol{sch}(R) = \{A_1, \ldots, A_m\}$. Moreover, let $U[\$t]$ be a parameterized relation with $\boldsymbol{sch}(U[\$t]) = \{A_1, \ldots, A_m, A'_1, \ldots, A'_m\}$ and $\boldsymbol{sch}(\$t) = \boldsymbol{sch}(Q)$. Finally, suppose that, for every $\$t \in Q$, the update of R defined by $U[\$t]$ is well-defined (cf. Definition 1).*

The rewrite rule in Fig. 1 is correct (i.e., the update resulting from the rewrite rule application has the same effect on the database as the loop), if the following conditions are fulfilled.

1. *In V, the functional dependency $V.\bar{A} \to V.\bar{A}'$ holds, i.e, the update of R by V is well-defined.*
2. *The relation $\rho_{\bar{A} \leftarrow \bar{A}'}(\pi_{\bar{A}'}(V)) \bowtie \pi_{\bar{A},\bar{A}'}(V)$ contains only tuples which fulfill the selection criterion $\sigma_{\bar{A}=\bar{A}'}$.*
3. *The relation R is not used in the definition of $U[\$t]$, i.e., $U[\$t]$ is not modified by the update.*

Remark 2. The second condition in the above theorem reads as follows: Consider all tuples in $\pi_{\bar{A}'}(V)$. They constitute a superset of the values that may possibly occur as the result value of some update step. The renaming $\rho_{\bar{A} \leftarrow \bar{A}'}$ and the join with $\pi_{\bar{A},\bar{A}'}(V)$ computes the result value (for arbitrary tuple $t \in Q$) if the update is applied to the same row of R again. The second condition thus requires that applying the update again must not alter the value anymore. In other words, the second condition imposes a strong kind of idempotence, i.e., if r' is the result obtained from updating $r \in R$ in the loop iteration according to some $t \in Q$, then the update of r' for any tuple $t' \in Q$ must not alter r'. Many real-world updates are idempotent: for instance, the commands setting attributes equal to constants, or looking up new values using join expressions, provided that the attributes used for the look-up are not affected by the update (cf. Example 3).

The third condition above means that we are considering loops with updates defined by relations U whose value is not modified inside the loop. Note that this restriction is quite realistic since otherwise the semantics of the loop might easily depend on the concrete order in which the tuples t of the "outer relation" Q are processed.

Note that if the update relation $U[\$t](\bar{A}, \bar{A}')$ is such that all attributes in $\bar{A}$ are either equal to the corresponding attributes in $\bar{A}'$ or not bound in the body of U, the second condition is fulfilled trivially (cf. Example 3). If also the first condition of updates to be well-defined is enforced by the schema constraints (as

described in Remark 1), then the applicability of the transformation in Fig. 1 can be checked statically, that is, without inspecting the actual instance.

Proof (Theorem 1). We first introduce some useful notation: Suppose that a tuple $r \in R$ is affected by the update in the i-th iteration of the loop, i.e., $r \in R$ is replaced by some tuple r'. Of course, it may happen that this result tuple r' itself is affected by the update in the j-th iteration of the loop with $j > i$. For this proof, it is convenient to think of the tuples $r \in R$ as equipped with an additional attribute *id*, which serves as a unique identifier of the tuples and which is never altered by any update. Hence, by identifying every tuple $r \in R$ with its unique *id*, we may distinguish between a tuple $id(r)$ and its value r. In particular, the updates only change the values of the tuples in R, while R contains always the same set of tuples.

Now let $T = \{t_1, \ldots, t_n\}$ denote the tuples in Q, s.t. the loop processes the tuples in Q in this (arbitrarily chosen) order. For every $i \in \{1, \ldots, n\}$, let Q_i and V_i be defined as $Q_i = \{t_1, \ldots, t_i\}$ and $V_i = \bigcup_{t \in Q_i} U[t]$. Clearly, it suffices to prove the following claim in order to prove the theorem:

Claim A. For every $i \in \{1, \ldots, n\}$, the update defined by the loop

$$\text{for } (\$t \text{ in } Q_i)\{\text{update R set } \bar{A} = U[\$t].\bar{A}' \text{ from } U[\$t] \text{ where } R.\bar{A} = U[\$t].\bar{A}\};$$

has the same effect on the database as the update

$$\text{update R set } \bar{A} = V_i.\bar{A}' \text{ from } V_i \text{ where } R.\bar{A} = V_i.\bar{A};$$

We proceed by induction on i with $i \in \{1, \ldots, n\}$:

"$i = 1$" In this case, we have $Q_1 = \{t_1\}$. Thus, the above for-loop is iterated exactly once and the corresponding update of R is defined by $U[t_1]$. On the other hand, we have $V_1 = U[t_1]$. Hence, the update defined by V_1 is precisely the same as the update in the (single iteration of) the loop.

"$(i-1) \to i$" By definition, $Q_i = Q_{i-1} \cup \{t_i\}$ and $V_i = V_{i-1} \cup U[t_i]$. We first show that the tuples of R affected by the first i iterations of the above loop coincide with tuples of R affected by the unnested update defined by V_i. In the second step, we will then show that the affected tuples of R are mapped to the same value by the loop-updates and by the unnested update.

Let $r \in R$ with identifier *id*. We observe the following equivalences: r is affected by an update in the first i iterations of the loop $\Leftrightarrow$ there exists a $j \leq i$ and a tuple $u \in U[t_j]$, s.t. $r.\bar{A} = u.\bar{A}$ holds $\Leftrightarrow$ r is affected by the update defined by $V_i = \bigcup_{t \in Q_i} U[t]$.

As for the value of the tuple $r \in R$ with identifier *id* after the i iterations of the for-loop respectively after the update defined by V_i, we distinguish 3 cases:

Case 1. Suppose that r is affected by the first $i-1$ iterations of the loop but the resulting tuple r' (which still has the same identifier *id*) is not affected by the i-th iteration. By the induction hypothesis, the updates carried out by the first $i-1$ loop iterations and the update defined by V_{i-1} have the same effect on r, namely they both modify r to r'. By assumption, this value is unchanged

in the i-th loop iteration. On the other hand, since $V_{i-1} \subseteq V_i$ and, by condition 1 of the theorem, the updated defined by V (and, therefore also by $V_i \subseteq V$) is well-defined. Hence, the update defined by V_i has the same effect on r as V_{i-1}.

Case 2. Suppose that r is affected by the i-th iteration of the loop for the first time. The update by the i-th loop iteration is defined by $U[t_i]$. On the other hand, $U[t_i] \subseteq V_i$ and, by condition 1 of the theorem, the updated defined by V (and, therefore also by $V_i \subseteq V$) is well-defined. Hence, both in the loop and in the unnested update, the tuple r is modified to the value r' according to the update defined by $U[t_i]$.

Case 3. Suppose that r is affected by the first $i-1$ iterations of the loop and the resulting tuple r' (which still has the same identifier *id*) is again affected by the i-th iteration. By the induction hypothesis, the updates carried out by the first $i-1$ loop iterations and the update defined by V_{i-1} have the same effect on r. Let the resulting value in both case be denoted as r'. Since $V_{i-1} \subseteq V_i$ and V_i is well-defined, the update defined by V_i also modifies r to r'. It remains to show that the i-th iteration of the loop does not alter r' anymore. Suppose that r' is modified to r'' by the update defined by $U[t_i]$. Clearly, $r' \in \pi_{\bar{A}'}(V)$. Moreover, $(r', r'') \in U[t_i] \subseteq V$ and, therefore, $(r', r'') \in \rho_{\bar{A} \leftarrow \bar{A}'}(\pi_{\bar{A}'}(V)) \bowtie \pi_{\bar{A},\bar{A}'}(V)$. Hence, $r' = r''$ by condition 3 of the theorem. □

If we want to apply the unnesting according to Theorem 1 to updates inside nested loops, we have to start from the innermost loop. Suppose that the nested loop looks as follows:

for ($\$t_1$ in Q_1) {
  for ($\$t_2$ in Q_2) {
    ...
      for ($\$t_n$ in Q_n) {update R
                  set $\bar{A} = U[\$t_1, \ldots, \$t_n].\bar{A}'$
                  from $U[\$t_1, \ldots, \$t_n]$
                  where $R.\bar{A} = U[\$t_1, \ldots, \$t_n].\bar{A}\}\ldots\}\}$;

In this case, $U[\$t_1, \ldots, \$t_n]$ plays the role of $U[\$t]$ in Theorem 1 and the conditions of Theorem 1 have to be fulfilled for all possible instantiations of the parameters $\$t_1, \ldots, \$t_{n-1}$ over the corresponding domains.

The following example illustrates the problems which could arise if no idempotence condition were required:

Example 4. Consider the program

for ($\$t$ in Q){update R set $A = U[\$t].A'$ from $U[\$t]$ where $R.A = U[\$t].A$};

such that R, Q, and $U[\$t]$ are relations with $\mathbf{sch}(R) = \{A\}$, $\mathbf{sch}(Q) = \{B_1, B_2\}$, and $\mathbf{sch}(U[\$t]) = \{A, A'\}$. Suppose that the update relation $U[\$t]$, which is parameterized by the tuple $\$t$, is defined as follows:

$U[\$t](A, A')$:- $A = \$t.B_1$, $A' = \$t.B_2$.

In other words, we consider a loop which is controlled by the tuples of Q, s.t. each tuple $t \in Q$ defines an update on R, namely if $t.B_1$ coincides with some entry $(A) \in R$, then (A) is replaced by $t.B_2$.

Table 1. Non-idempotence of updates

$\mathbf{R.A_1}$	$\mathbf{Q.B_1}$	$\mathbf{Q.B_2}$	$\mathbf{Q'.B_1}$	$\mathbf{Q'.B_2}$
1	1	2	1	2
2	2	3	2	2

First, suppose that the relations R and Q are given in Table 1. In this case, the result of the update loop depends on the order in which the elements of Q are processed: if the tuple $(1, 2) \in Q$ is chosen first, then both tuples in R are updated to (3) (the first tuple of R is processed by each iteration: 1 is replaced with 2 and then further replaced with 3 at the second iteration). On the other hand, if the tuple $(2, 3) \in Q$ is processed first by the loop, then R is updated to $\{(2), (3)\}$ by the entire loop.

Clearly, a loop whose result depends on the order in which the tuples of Q are selected is usually not desired. In this case, condition 2 of Theorem 1 is violated and, hence, the rewriting of Fig. 1 is not allowed. Indeed, condition 2 requires that the relation $(\rho_{B_1 \leftarrow B_2}(\pi_{B_2}(Q)) \bowtie_{B_1=B_2} Q)$ consist of tuples with two equal columns, which is not the case.

Now suppose that we use the relation Q' instead of Q. Then the condition 2 of Theorem 1 is satisfied. Indeed, with any order of selecting the tuples of Q' in the loop, R gets modified to $\{(2), (2)\}$. □

As could be seen in the previous example, the violation of condition 2 of Theorem 1 may indicate an undesired behavior of the loop. However, a non-idempotent behavior of an update inside a loop is not always undesired. For instance, consider the following variation of the update in Example 3 increasing the salary of each employee who has been participating in a successfully finished project.

Example 5. Consider the schema of Example 3 extended with the relations Project and EmployeeProject with $\mathbf{sch}(\mathsf{EmployeeProject}) = \{\mathsf{empl\_id}, \mathsf{proj\_id}\}$ and $\mathbf{sch}(\mathsf{Project}) = \{\mathsf{proj\_id}, \mathsf{status}\}$. Suppose that the employee's compensation grows depending on the number of successful projects she has been working in. For instance, the following statement can be used to update the *Employee* table to reflect such a policy:

$$Q(Empl\_id, Proj\_id) \text{:- } \mathsf{EmployeeProject}(Empl\_id, Proj\_id),\ \mathsf{Project}(Proj\_id, \text{'success'})$$

$$U(Empl\_id, Dept\_id, Base\_sal, Comp, Empl\_id, Dept\_id, Base\_sal, Comp')[\$t] \text{ :-}\ Empl\_id = \$t.\mathsf{empl\_id},\ \mathsf{Department}(Dept\_id, Bonus),\ Comp' = Comp \cdot (1 + Bonus)$$

for ($\$t$ in Q)$\{\mathrm{up}_1; \ldots \mathrm{up}_n\}$;
$\vdash$ for ($\$t$ in Q)$\{\mathrm{up}_{j_1}; \ldots \mathrm{up}_{j_\alpha}\}$; for ($\$t$ in Q)$\{\mathrm{up}_{j_{\alpha+1}}; \ldots \mathrm{up}_{j_n}\}$;
s.t. up_i denotes the update of R_i by $U_i[\$t]$, i.e., up_i is of the form update R_i set $\bar{A}_i = U_i[\$t].\bar{A}'_i$ from $U_i[\$t]$ where $R_i.\bar{A}_i = U_i[\$t].\bar{A}_i$;

Fig. 2. Unnesting update programs

for($\$t$ in Q)
update Employee set compensation = $U[\$t]$.compensation′ from $U[\$t]$
where $U[\$t]$.empl_id = Employee.empl_id and
$U[\$t]$.dept_id = Employee.dept_id and
$U[\$t]$.base_salary = Employee.base_salary and
$U[\$t]$.compensation = Employee.compensation

The update is well-defined but not idempotent: the incremented compensation depends on the previous compensation. The restriction of Theorem 1 is not desirable here. □

Different special cases like this leave a space for refining the preconditions for loop elimination. E.g., the update predicate can be iterated some reasonable number of times to check if the update becomes deterministic at some point reachable by the update loop.

4 Loops with More Than One Update

In this section, we consider the case of two (or more updates) inside a loop. If these updates operate on different relations R_1 and R_2, then the loop can obviously be decomposed into two loops with a single update inside. More generally, we define the following rewrite rule.

Theorem 2. *Let Q, $R_1, \ldots, R_n$ be relations with $\boldsymbol{sch}(Q) = \{B_1, \ldots, B_\ell\}$ and $\boldsymbol{sch}(R_i) = \bar{A}_i = \{A_{i1}, \ldots, A_{im_i}\}$ for $i \in \{1, \ldots, n\}$, and let let $U_1[\$t], \ldots, U_n[\$t]$ be parameterized relations with $\boldsymbol{sch}(U_i[\$t]) = \{A_{i1}, \ldots, A_{im_i}, A'_{i1}, \ldots, A'_{im_i}\}$ and $\boldsymbol{sch}(\$t) = \boldsymbol{sch}(Q)$. Moreover, suppose that, for every $\$t \in Q$, the update of R_i defined by $U_i[\$t]$ is well-defined (cf. Definition 1).*

The rewrite rule in Fig. 2 is correct (i.e., the two loops resulting from the rewrite rule application have the same effect on the database as the original loop), if the following conditions are fulfilled.

1. *The set $\{1, \ldots, n\}$ is partitioned into two sets $J_1 = \{j_1, \ldots, j_\alpha\}$ and $J_2 = \{j_{\alpha+1}, \ldots, j_n\}$, s.t. the two sequences of indices $(j_1, \ldots, j_\alpha)$ and $(j_{\alpha+1}, \ldots, j_n)$ are arranged in increasing order.*
2. *$\{R_s \mid s \in J_1\}$ and $\{R_s \mid s \in J_2\}$ are disjoint.*

Proof. Let $T = \{t_1, \ldots, t_N\}$ denote the tuples in Q, s.t. the loop processes the tuples in Q in this (arbitrarily chosen) order. For every $k \in \{1, \ldots, N\}$, let $T_k = \{t_1, \ldots, t_k\}$. We claim that, for every $k \in \{1, \ldots, N\}$ the following rewriting is correct:

$$\begin{array}{l}\text{for } (\$t \text{ in } T_k)\{\mathrm{up}_1; \ldots \mathrm{up}_n\};\\ \vdash \quad \text{for } (\$t \text{ in } T_k)\{\mathrm{up}_{j_1}; \ldots \mathrm{up}_{j_\alpha}\};\ \text{for } (\$t \text{ in } T_k)\{\mathrm{up}_{j_{\alpha+1}}; \ldots \mathrm{up}_{j_n}\};\end{array}$$

The correctness of this decomposition of the loop into two loops can be proved by an easy induction argument which uses the facts that the relations $U_1, \ldots, U_n$ are never modified inside these loops and the updates in the two resulting loops operate on different relations R_s with $s \in J_1$ and $R_{s'}$ with $s' \in J_2$. Hence, there is no interdependence between the updates in the two resulting loops. □

From now on, we may concentrate on the case that all updates in a loop operate on the same relation R. Below we define a rewrite rule for contracting two updates of the same relation R to a single update. By repeating this rewrite step, any number of updates of the same relation R can be rewritten to a single update of R.

Theorem 3. *Let R, U_1, and U_2 be relations with $\boldsymbol{sch}(R) = \{A_1, \ldots, A_m\}$ and $\boldsymbol{sch}(U_i) = \{A_1, \ldots, A_m, A'_1, \ldots, A'_m\}$ for $i \in \{1, 2\}$ and suppose that the update defined by each U_i is well-defined. Moreover, let U'_i be defined as follows:*

$$\begin{array}{l}U'_i(X_1, \ldots, X_m, X'_1, \ldots, X'_m) \text{ :- } U_i(X_1, \ldots, X_m, X'_1, \ldots, X'_m).\\ U'_i(X_1, \ldots, X_m, X_1, \ldots, X_m) \text{ :- } R(X_1, \ldots, X_m),\\ \qquad\qquad\qquad\qquad\quad \textit{not } U_i(X_1, \ldots, X_m, \_, \ldots, \_).\\ U'_i(X_1, \ldots, X_m, X_1, \ldots, X_m) \text{ :- } U_1(\_, \ldots, \_, X_1, \ldots, X_m),\\ \qquad\qquad\qquad\qquad\quad \textit{not } U_1(X_1, \ldots, X_m, \_, \ldots, \_).\\ U'_i(X_1, \ldots, X_m, X_1, \ldots, X_m) \text{ :- } U_2(\_, \ldots, \_, X_1, \ldots, X_m),\\ \qquad\qquad\qquad\qquad\quad \textit{not } U_2(X_1, \ldots, X_m, \_, \ldots, \_).\end{array}$$

Finally, we define V with $\boldsymbol{sch}(V) = \{A_1, \ldots, A_m, A'_1, \ldots, A'_m\}$ as follows:

$$\begin{array}{l}V(X_1, \ldots, X_m, X'_1, \ldots, X'_m) \text{ :- } U'_1(X_1, \ldots, X_m, Y_1, \ldots, Y_m),\\ \qquad\qquad\qquad\qquad\qquad\quad U'_2(Y_1, \ldots, Y_m, X'_1, \ldots, X'_m).\end{array}$$

Then the rewrite rule in Fig. 3 is correct, i.e., the update of R defined by V is also well-defined and has the same effect on the database as the two successive updates of R by U_1 and U_2.

Remark 3. Note that there are two possibilities why the update of a relation R defined by some relation U leaves a value combination $(a_1, \ldots, a_m)$ of the attributes $(A_1, \ldots, A_m)$ unchanged: either U does not contain a row, s.t. the first m columns

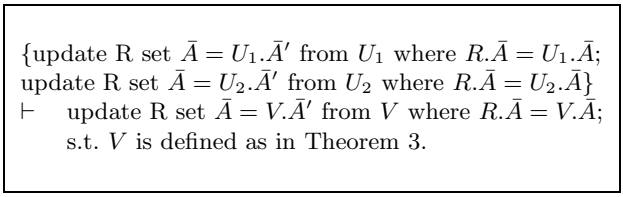

Fig. 3. Contracting two updates

coincide with $(a_1, \ldots, a_m)$; or U contains the row $(a_1, \ldots, a_m, a_1, \ldots, a_m)$. Intuitively, the latter case makes the identity mapping for the tuple $(a_1, \ldots, a_m)$ in R explicit. The intuition of each relation U_i' in the above theorem is that it defines exactly the same update of R as U_i. The only difference between U_i' and U_i is that U_i' makes all possible identity mappings explicit.

Proof. Let r be an arbitrary tuple in r and suppose that r is modified to r' by the update defined by U_1 (of course, $r' = r$ if r is not affected by this update). Moreover, let r' be further modified to r'' by the update defined by U_2. Then either $r \notin \pi_{\bar{A}}(U_1)$ or $(r, r') \in U_1$. In either case, $(r, r') \in U_1'$. Likewise, we may conclude that $(r', r'') \in U_2'$ holds. Hence, also $(r, r'') \in U_1' \bowtie U_2' = V$ holds. Note that the value of r' is uniquely determined by r. This is due to the definition of U_1' and to the fact that U_1 is well-defined. Likewise, the value of r'' is uniquely determined by r'. Hence, the update defined by V is well-defined. Moreover, it indeed modifies r to r''. □

In total, we define the following algorithm for unnesting updates in for-loops:

1. In case of nested for-loops, start with the innermost loop.
2. If a loop contains several updates affecting more than one relation, then replace the for-loop by several successive for-loops each updating a single relation (by iteratively applying the rule of Fig. 2).
3. If a loop contains several updates which all affect the same relation, then replace this sequence of updates by a single update (by iteratively applying the rule of Fig. 3).
4. Replace a loop with a single update by an update without loop (by applying the rule of Fig. 1).

From a program optimization point of view, also partial unnesting via our techniques may lead to much more efficient queries – even if complete unnesting is not always possible (due to the conditions which are required for our transformation rules to be correct).

5 Conclusion

We have considered the problem of unnesting relational updates with cursors and replacing them with simpler, purely relational update expressions. The full set of our rewrite rules can handle loops with one or multiple update statements.

Unnesting and loop elimination can drastically simplify the database program, making it truly declarative and thus more readable and accessible for optimization through appropriate components of the database engine.

Our technique crucially relies on the idempotence of the update operation. Reasonable in most cases, in some situations this requirement can be too restrictive, as discussed in Section 3 (see Example 5). More fine-grained optimization techniques of update loops, relaxing the idempotence requirement where appropriate, as well as more elaborate techniques of splitting loops with multiple updates are left for future work.

Acknowledgements. The research of V. Savenkov and R. Pichler is supported by the Austrian Science Fund (FWF): P25207-N23. The work of C. Koch is supported by Grant 279804 of the European Research Council.

References

1. Antova, L., Koch, C.: On APIs for probabilistic databases. In: QDB/MUD, pp. 41–56 (2008)
2. de Kruijf, M.A., Sankaralingam, K., Jha, S.: Static analysis and compiler design for idempotent processing. SIGPLAN Not. 47(6), 475–486 (2012)
3. Gluche, D., Grust, T., Mainberger, C., Scholl, M.H.: Incremental updates for materialized OQL views. In: Bry, F. (ed.) DOOD 1997. LNCS, vol. 1341, pp. 52–66. Springer, Heidelberg (1997)
4. Gupta, A., Mumick, I.S. (eds.): Materialized views: techniques, implementations, and applications. MIT Press, Cambridge (1999)
5. Lieuwen, D.F., DeWitt, D.J.: A transformation-based approach to optimizing loops in database programming languages. SIGMOD Rec. 21(2), 91–100 (1992)
6. Ramalingam, G., Vaswani, K.: Fault tolerance via idempotence. In: Proc. of POPL 2013, pp. 249–262. ACM, New York (2013)