Admissible Record-Oriented Evaluation Plans for Declarative Updates

Gisela Fischer
Dresdner Bank AG, 60301 Frankfurt a.M., Germany
E-Mail: Gisela.Fischer@dresdner-bank.com

Karl Aberer
GMD-IPSI, Dolivostrasse 15, 64293 Darmstadt, Germany
E-Mail: aberer@darmstadt.gmd.de

Abstract

Efficient evaluation strategies for declarative updates have rarely been investigated. Due to possible dependencies between the resulting database state and the order in which records (objects) are processed, usually declarative updates are evaluated in a set-oriented way in order to ensure a deterministic evaluation. In this paper, we show that such dependencies can be detected by exploiting knowledge about conflicts between the operations that are used to access the database during the update evaluation. Thus most declarative updates can also be evaluated deterministically, and in some cases more efficiently, in a record-oriented way. We show that some of the detected conflicts can be relaxed or even be ignored, while a deterministic evaluation can still be guaranteed.

1 Introduction

Different approaches for optimizing and evaluating declarative queries have been proposed so far (comprehensive overviews can be found in [10] and [6]). Many query languages like SQL[9], QUEL[15] or POSTQUEL[13] also provide means for a declarative specification of database updates. However, particular strategies for the optimization and especially for the efficient evaluation of declarative updates have hardly been investigated. Most of the optimization strategies that have been developed for queries, e.g., algebraic optimization using equivalences based on heuristics, can also be applied to declarative updates with minor modifications. Unfortunately, this is not possible for the corresponding evaluation strategies, since in some cases the result of a declarative update depends on the order in which records (objects) are processed, as we will show later.

We assume that the evaluation of queries and declarative updates is realized by executing several algorithms which correspond to, e.g., join, selection and projection. Each algorithm produces a result and/or consumes the result(s) of the previously executed algorithm(s). One can distinguish the following two basic processing strategies for these algorithms:

- *set-at-a-time (set-oriented):* An algorithm processes sets of records. The result set(s) of the previous algorithm(s) is (are) processed completely within an algorithm before its own result set is propagated to the next algorithm.
- *record-at-a-time (record-oriented):* An algorithm processes a single record. A result record computed from the input record(s) is immediately propagated to the next algorithm.

If additional algorithms for switching between set-oriented and record-oriented processing and vice versa exist, both set-oriented and record-oriented algorithms can be combined in the evaluation. If only set-oriented (record-oriented) algorithms are used, we say that the evaluation is set-oriented (record-oriented), or the query/update is evaluated in a set-oriented(record-oriented) way. Otherwise we say the evaluation is *mixed*. In certain cases only the

set-oriented strategy is applicable, e.g, when aggregations have to be computed, sorting is used, or for the enforcement of certain semantic integrity constraints for declarative updates. If record-at-a-time is applicable, a record-oriented evaluation offers several advantages. First of all, it avoids the computation of large intermediate results which probably have to be stored on secondary storage because of main memory limitations. Second, the evaluation can be parallelized using *pipelining* (e.g. see [6][14][19]). Third and last, if the database buffer is too small to hold all records which have to be processed, a record-oriented evaluation can avoid multiple replacing and subsequent reloading of the same records.

For queries, set-oriented, record-oriented and mixed evaluation yield the same result. By contrast, evaluating a declarative update in a record-oriented can lead to a different database state than evaluating it in a set-oriented way. In fact, the former may even lead to a non-deterministic evaluation. This problem has been identified for updates in relational databases as the following examples show.

Example 1 (known as the Halloween problem [6])
The relation *Employees* is given as follows:

name	**salary**	**manager**
Smith	35.000	Smith
Brown	32.000	Smith
Jones	30.000	Brown
Miller	27.000	Jones

The following update is posed against *Employees* : "Raise the salary of all employees who earn more than $30.000 by 10%."

Assume that an index whose entries are sorted in ascending order is defined on the attribute $salary$. If we evaluate this declarative update in a set-oriented way, first all relevant employees are selected, then their salary is increased (in this example the employees Smith and Brown), and the index is updated. - For a record-oriented evaluation, we assume that each relevant record is selected separately using the index defined on $salary$, and that index entries are read in ascending order. In order to keep the index consistent with the database state, changes of salary values are immediately stored within the corresponding index entries. Then, instead of a single salary increase, Smith and Brown get an infinite number of raises since their index entries are swapped with each update, and the records are selected and updated again.

Example 2 (given in a similar form in [15])
The relation *Employees* is given as in Example 1. Now all employees whose managers earn at least $33.000 should get a salary increase by 10%.

With a set-oriented evaluation, obviously only Smith's and Brown's salary is increased. But if we follow the record-oriented strategy and examine records in the order given in Example 1, the following happens:

- Smith meets the selection condition; his salary is increased to $38.500.
- Brown also meets the selection condition; his salary is increased to $35.200.
- Jones now also meets the selection conditions since his manager's (= Brown's) salary is greater that $33.000. His salary is changed to $33.000.
- Because of the same reason, Miller also gets a salary increase.

In this paper we consider only one special type of declarative updates, namely the application of an update operation to a set of objects that is determined by a query. Furthermore, we assume that the declarative updates we investigate can be evaluated deterministically, i.e., they give a unique result when evaluated using the set-oriented strategy ([2] and [11] describe approaches for detecting and handling declarative updates which cannot be evaluated deterministically).

A non-deterministic evaluation of a declarative update is caused by conflicts between the database access operations that are executed during the evaluation. In this paper we investigate whether a record-oriented evaluation is still correct, although conflicts exist. First, we introduce a query processing model for an object-oriented DBMS as a basis for the investigation of declarative updates. We think that the development of concepts for dealing with declarative updates is even more important in the context of object-oriented databases, since updates can be performed by database methods, which may occur (at least syntactically) anywhere in queries formulated in a declarative object-oriented query language. Then we develop a formal model which is based on tracing the database access operations during query evaluation. Since our concept is based on an object-oriented data model, we represent all database access operations as system-defined and user-defined methods, respectively. The former represent, e.g., scans on class extensions or index scans (then the respective class object is the receiver)[1], while the latter are application-specific methods defined in a database schema. The trace model is particularly suited to analyze conflicts that may occur during the evaluation of declarative updates. For conflict detection, we exploit conflict specifications that are given together with the methods. We identify two frequently occurring non-trivial cases where declarative updates can be evaluated deterministically in a record-oriented way, although conflicting methods occur within the evaluation. The concept we describe in this paper allows to consider alternative, possibly more efficient, evaluation strategies for declarative updates and thus can be regarded as a special optimization technique. To the best of our knowledge, this is the first paper that investigates this optimization potential for declarative updates.

The remainder of the paper is organized as follows. Section 2 gives an overview on related work. In Section 3 we briefly describe the logical query algebra declarative updates are mapped to, introduce the physical query algebra which is used to build up evaluation plans, and illustrate how its operators are mapped to concrete algorithms. We examine the application of the record-oriented strategy to declarative updates in Section 4, and show that some of the detected conflicts can be relaxed or even ignored, while a deterministic evaluation can still be guaranteed. In Section 5, perspectives for the applicability of our concept within a rule-based query optimization framework are sketched and some performance considerations are given. Section 6 concludes the paper.

2 Related Work

About twenty years ago, non-determinism in evaluating declarative updates was first recognized. As a consequence, declarative updates in, e.g., INGRES [16] and POSTGRES [13] are only evaluated in a set-oriented way. This restriction is mostly used for a straightforward solution to this problem (e.g., [6]). In fact, it does not even solve the problem in general, as it is shown in two recent publications that deal with non-deterministic evaluation of declarative updates explicitly [11][2]. Both concentrate on the problem of applying a sequence of update operations, or an update method, resp., to a set of receivers. Conflicts between update operations/update methods that are applied to different receivers are examined. In [11] state-independent conflict specifications, which are also utilized for concurrency control, are used to detect non-determinism. If a conflict occurs at run time (which is detected by the concurrency control), the execution of the declarative update is either stopped and rolled back, or the execution is continued with non-deterministic semantics. - In [2] a more theoretical approach is introduced. The authors concentrate on the analysis of declarative updates on the basis of so-called schema colorings which are used to classify all update methods in a schema according to their update behavior (update, creation and deletion of objects). They show that these colorings can be used to decide whether the application of an update method to a set of receivers leads to a non-deterministic evaluation or not. It is proven that for a so-called key-order independent method, a sequential application of the method to a set of receivers is equal to a parallel application.

In the following we concentrate on the examination of conflicts between update and read-only methods. As mentioned before, we assume that the declarative updates we look at can be evaluated deterministically in a set-oriented way. This can be proved by using the techniques introduced in [11] and [2].

[1] Of course, a scan on the extension of a class does not have to be implemented in the form of a method; we just represent it as one in order to have a uniform model for all database access operations.

3 The Query Processing Model

In general, declarative updates are formulated using a declarative language. We do not rely on a specific language, but assume that a declarative language like ODMG-93 OQL [4] or VQL [1][5] is used. The main requirement is that methods, including update methods, can be used in the formulation of queries and declarative updates, provided that the semantic correctness of the query/update is given.

We follow an algebraic approach with the distinction of a logical and a physical query algebra as introduced in [7]. Queries and declarative updates are translated into logical algebra expressions. The query optimizer transforms these logical expressions, e.g., pushs selections down as far as possible, maps logical to physical algebra expressions using so-called *implementation rules*, and chooses the cheapest physical expression as the evaluation plan.

3.1 Logical Query Algebra

The logical algebra we use in this paper has been introduced in [1]. It is not complete with regard to query languages like ODMG-93 OQL. For the sake of simplicity the expressiveness of the algebra is restricted to the class of declarative updates to which the results of this paper apply.

The operators of the logical as well as the physical query algebra are applied to complex values of type $\{[a_1 : D_1, \ldots, a_n : D_n]\}$ where $D_1, \ldots, Dn$ are complex data types. We assume that the record components are unordered. Operator arguments of this type are denoted by S. The operator parameters are enclosed in $<>$. We define $Ref(S) := \{a_1, \ldots, a_n\}$ for $Type(S) = \{[a_1 : D_1, \ldots, a_n : D_n]\}$ and refer to $a_1, \ldots, a_n$ as the *references* of S. In the following p and m denote property and method identifiers, respectively. The invocation of a method m with parameters $p_1, \ldots, p_k$ and receiver o is represented by $o \rightarrow m(p_1, \ldots, p_k)$.

The logical algebra consists of the operators *select, project, join, natural_join, union* and $diff$ as well as the operators *get, map_const, map_op, map* and $flat$. The former are defined in analogy to the commonly known operators from relational algebras. The latter are provided for representing class extensions, constants, operations on the built-in data types and method invocations, respectively ($flat$ flattens set-valued methods results).

The declarative updates we consider in this paper are restricted to the form

$select\, rec \rightarrow m(p_1, \ldots, p_k)$
$from\, x_1\, in\, e_1, \ldots, x_n\, in\, e_n\, where\, pred$

(given in the language of ODMG-93 OQL), where m is an update method, and the other expressions for the method receiver rec, the method parameters $p_1, \ldots, p_k$, the domains $e_1, \ldots, e_n$ for query variables, and the query predicate $pred$ contain exclusively invocations of read-only methods.

Declarative updates are then mapped to a logical algebra expression of the form

$map < a, m, a_{rec}, < a_{p_1}, \ldots, a_{p_k} >> (E)$

where E is the logical algebra expression which represents the $from$ and the $where$ clause, and m is the update method which is invoked in the $select$ clause.

Example 3 We assume that a class Emp with the method $raiseSalary(p : INT) : BOOL$ is defined in analogy to the relation $Employees$. The declarative update from Example 1 can be formulated as

$select\, e \rightarrow raiseSalary(10)$
$from\, e\, in\, Emp$
$where\, e.salary > 30.000$

Note that $e.salary$ is a short-hand notation for the invocation of a system-defined method $salary()$ which is provided for reading property values. The representation in the logical algebra is then given by

$map < a_5, raiseSalary, a_1, < a_4 >> ($
  $map\_const < a_4, 10 > ($
    $select < a_2, >, a_3 > ($
      $map\_const < a_3, 30.000 > ($
        $map < a_2, salary, a_1 > ($
          $get < a_1, Emp >)))))$

3.2 Physical Query Algebra

The physical query algebra contains operators which are associated with certain cost functions and evaluation algorithms. For each operator in the logical query algebra, there exists (at least) one corresponding operator of the physical query algebra. For some logical operators, there exist alternative physical operators, e.g., for $join$ we provide the physical operators $nested\_loop\_join, hash\_join$ and $merge\_join$ which represent different join algorithms. Additionally, the physical algebra contains the operators $sort$ for sorting the input sets of a merge join, the operator $select\_index < a, C, p, v >$ for selecting the instances of the class C using the index defined on property p with value v, and the operator $collect$ to accumulate (intermediate) result sets.

The optimizer generates several physical algebra expressions which we refer to as *evaluation plans* (EP). The overall cheapest EP is then chosen to evaluate the query/declarative update. We refer to an EP P where each operator op is covered by a $collect$ operator, i.e., $P = collect(op_1(\ldots collect(op_k)\ldots)$ as a set-oriented EP. The corresponding record-oriented EP $P' = op_1(\ldots(op_k)\ldots)$ is obtained from P by removing the $collect$ operators. If some, but not all operators in an EP are covered by a $collect$ operator, we refer to the EP as a mixed EP.

In order to analyze the correct execution of declarative updates, we give the concrete algorithms for some of the physical algebra operators. For each physical operator except $sort$, we provide algorithms for a record-oriented evaluation. The corresponding set-oriented processing strategy for an operator op can then be obtained by covering the operator with a $collect$ operator, i.e., $collect(op)$. For $sort$, a set-oriented algorithm is provided.

Since record-oriented algorithms can be thought of as iterators on streams of records [6], we provide for each operator algorithms $OPEN$ and $NEXT$, for opening the stream and obtaining the next element of the stream, respectively. In the following algorithms P, P_1 and P_2 are the operator arguments. Θ denotes a boolean predicate on built-in data types, C is a class name, c is a constant, and o is the OID of an instance of C. S and U denote sets, while s, t and u denote records. The methods $open\_scan(a)$ and $scan(a)$ are defined for class objects and realize a scan on the class extension. $scan(a)$ returns a record of the form $[a : o]$ where o is the OID of an instance of a class C which is the receiver of this method. The methods $open\_scan\_index(a, p, v)$ and $scan\_index(a, p, v)$ are defined in analogy to scan the index which is defined for the class C on property p with value v. The operation $concat(s, t)$ concatenates records s and t, and returns the concatenated record. The operations $open(S)$ and $next(S)$ are used to iterate on the set S. As examples for the implementation of $OPEN$ and $NEXT$, we give the corresponding algorithms for the operators *get, map, nested_loop_join* and $collect$. For $select\_index$, the algorithms can be inferred from the corresponding ones given for the operator get by replacing $open\_scan(a)$ through $open\_scan\_index(a, p, v)$ and $scan(a)$ through $scan\_index(a, p, v)$, respectively.

$OPEN(get < a, C >);$
$\quad \{C \rightarrow open\_scan(a);\}$

$NEXT(get < a, C >) : s;$
$\quad \{s := C \rightarrow scan(a);$
$\quad IF\ (s! = NULL)$
$\quad THEN\ RETURN\ s\ ELSE\ RETURN\ NULL;\}$

$OPEN(map < a, m, a_1, < a_2, \ldots, a_k >> (P));$
$\quad \{OPEN(P);\}$

$NEXT(map < a, m, a_1, < a_2, \ldots, a_k >> (P)) : s;$
$\quad \{t := NEXT(P);$
$\quad IF\ (t! = NULL)\ THEN\ RETURN$
$\quad\quad concat([a : t.a_1 \rightarrow m(t.a_2, \ldots, t.a_k)], t)$
$\quad ELSE\ RETURN\ NULL;\}$

$OPEN(nested\_loop\_join < a_1, \Theta, a_{i+1} > (P_1, P_2));$
$\{U := \{\};\ OPEN(P_2);\ u := NEXT(P_2);$
$WHILE\ (u! = NULL)$
$\{U := U\ UNION\ \{u\};\ u := NEXT(P_2);\}$
$OPEN(P_1);\ t := NEXT(P_1);\ open(U);$
$u := next(U)\}$

$NEXT(nested\_loop\_join < a_1, \Theta, a_{i+1} > (P_1, P_2)) : s;$
$\{s := NULL;$
$WHILE\ (s = NULL)\ AND\ (t! = NULL)$
$\{IF\ (u! = NULL)\ THEN$
$\{WHILE\ (u! = NULL)\ AND\ (s = NULL)$
$IF\ (t.a_1 \Theta u.a_{i+1})\ THEN\ s := concat(t, u);$
$u := next(U);\}$
$ELSE\ \{u := open(U);\ u := next(U);$
$t := NEXT(P_1);\}\}$
$RETURN\ s;\}$

$OPEN(collect(P));$
$\{OPEN(P);\ S := \{\};\ s := NEXT(P);$
$WHILE\ (s! = NULL)$
$\{S := S\ UNION\ \{s\};\ s := NEXT(P);\}$
$open(S);\}$

$NEXT(collect(P)) : s;$
$\{RETURN\ next(S);\}$

4 Record-Oriented Evaluation of Declarative Updates

Non-determinism in the evaluation of declarative updates is caused by conflicts between the methods that are executed during the evaluation of a record-oriented EP. In general, two methods a and b are said to commute if their execution order can be switched without causing any changes. a can be executed before as well as after b, and the results of both methods and of any subsequent method c, which is executed on a database state that has eventually been modified by a and b, do not change. Otherwise a and b are said to be in conflict.

We utilize so-called state-independent commutativity specifications, where only information about the method itself (i.e., its name) and its actual parameters which are known at compile time are used for a conflict test [18]. Note that since we do not consider information about the actual database state, on the one hand we can perform a conflict test at compile time. On the other hand, we can only detect possible conflicts which might, but do not necessarily occur.

Commutativity specifications have mainly been exploited in semantic concurrency control [3][12][18] to achieve a higher degree of parallelism. In this context conflict tests are performed at run time. It is assumed that methods which are invoked for different receiver objects commute (local atomicity property), since a method can only directly manipulate the state of its receiver. If conflicts occur due to nested method invocations, they are detected at run time [12]. In the case of declarative updates, a conflict test should preferably be performed at compile time, or at least before the evaluation starts, in order to avoid a non-deterministic evaluation or a rollback of all method invocations executed so far during the evaluation. Since conflicts due to nested method invocations cannot be detected then, methods invoked for different receivers cannot be considered as commuting by default, as it is done in semantic concurrency control. For our purposes, we extend the notion of commutativity with respect to the receiver objects of methods as follows:

Definition 1 (Total and partial commutativity)
Two methods m_1 and m_2 commute *totally* iff for any database state DB_0 the execution sequences $(o_1 \rightarrow m_1 :$

$r_1; o_2 \rightarrow m_2 : r_2)$ and $(o_2 \rightarrow m_2 : r_2; o_1 \rightarrow m_1 : r_1)$ lead to same resulting database state DB_1, and the results r_1 and r_2 of the two method invocations are the same. They commute *partially* if additionally for the receiver objects $o_1 \neq o_2$ holds.

4.1 Traces

To investigate conflicts between methods that are invoked during the evaluation, we need a precise description of the order of method invocations. For this purpose, we introduce the notion of *traces* of method invocations. In order to allow a recursive computation of traces for EPs, we also include the intermediately generated records that are passed on to the other operators during evaluation, since these records lead to further method invocations. The trace for an EP P displays all method invocations that are executed and all records that are generated, when for P the sequence of calls $OPEN(P); NEXT(P) : t_1; \ldots NEXT(P) : t_n; NEXT(P) : NULL;$ is performed. We define the trace of the execution of an EP P against a given database state as follows:

Definition 2 (TRACE)
The trace $TRACE(P)$ of an EP P is the sequence consisting of method invocation sequences $M_i, i = 0, \ldots, n+1$, records $t_i, i = 1, \ldots, n$, and the entry *opened*, denoted as

$$TRACE(P) =< M_0;\ opened;\ M_1;\ t_1;\ \ldots; M_n;\ t_n;\ M_{i+1} >$$

such that for the execution of $OPEN(P);\ NEXT(P) : t_1; \ldots NEXT(P) : t_n;\ NEXT(P) : NULL;$ all method invocations before the entry *opened* correspond to those executed in $OPEN(P)$, the sequence $M_i,\ i = 1..n$, corresponds to the method invocations executed in $NEXT(P) : t_i$, and the sequence M_{i+1} corresponds to the method invocations executed in $NEXT(P) : NULL$.

According to this definition, we can derive the traces for the physical algebra operators given in Section 3.2 from the corresponding algorithms.

$$TRACE(get < a, C >) :=$$
$$< C \rightarrow open\_scan(a);\ opened; C \rightarrow scan(a);\ s_1; \ldots; C \rightarrow scan(a);\ s_n; C \rightarrow scan(a); >$$

Let $TRACE(P) :=< M_0;\ opened;\ M_1;\ t_1; \ldots; M_n;\ t_n;\ M_{i+1} >$

$$TRACE(map < a, m, a_1, < a_2, \ldots, a_k >> (P)) :=< M_0;\ opened;\ M_1;\ t_1.a_1 \rightarrow m(t_1.a_2, \ldots, t_1.a_k);$$
$$s_1(t_1); \ldots; M_n; t_n.a_1 \rightarrow m(t_n.a_2, \ldots, t_n.a_k);\ s_n(t_n); M_{i+1} >$$

The expression $s(t)$ denotes a record s that is an extension of the record t, i.e., $\forall a \in Ref(t)$[2], $a \in Ref(s)$ and $t.a = s.a$. E.g., for *map*, t is extended by a component which holds the result of the method call.

$$TRACE(nested\_loop\_join < a_1, \Theta, a_{i+1}) > (P, P')) :=$$
$$< M'_0;\ M'_1;\ \ldots; M'_m;\ M'_{m+1};\ M_0;\ opened;$$
$$M_1;\ v_{1,1}(t_1, u_1)?; \ldots; v_{1,m}(t_1, u_m)?; \ldots; M_n; v_{n,1}(t_n, u_1)?; \ldots; v_{n,m}(t_n, u_m)?; M_{i+1} >$$

where $v_{i,j}(t_i, u_j)?$ means that the record $v_{i,j}$ is generated from t_i and u_j and may not be contained in the trace, and where $TRACE(P') =< M'_0;\ opened;\ M'_1;\ u_1; \ldots; M'_m;\ u_m;\ M'_{m+1} >$

$$TRACE(collect(P)) :=< M_0;\ M_1; \ldots; M_n;\ M_{i+1}; opened; t_1; \ldots; t_n >$$

The sequence of method invocations in the trace corresponds exactly to the sequence of method invocations in the execution of the EP, and the sequence of records in the trace corresponds exactly to the result computed by the EP.

Example 4 The set-oriented and the corresponding record-oriented EP for the declarative update described in Example 1 can be formulated as follows:

[2] For $t \in T = \{[a_1 : v_1, \ldots, a_n : v_n]\}$, $Ref(t) = Ref(T)$

$$
\begin{aligned}
P \;=\; & collect(map < a_3, raiseSalary, a_1, < a_2 >> (\\
& collect(map\_const < a_2, 10 > (\\
& \quad collect(select\_index < a_1, Emp, salary, 30.000 >)))))
\end{aligned}
$$

$$
\begin{aligned}
P' \;=\; & map < a_3, raiseSalary, a_1, < a_2 >> (\\
& map\_const < a_2, 10 > (\\
& \quad select\_index < a_1, Emp, salary, 30.000 >))
\end{aligned}
$$

The traces T and T' of P and P', respectively, are then

$$
\begin{aligned}
T \;=\; & TRACE(P) \;= \\
& < Emp \rightarrow open\_scan\_index(a_1, salary, 30.000); \\
& Emp \rightarrow scan\_index(a_1, salary, 30.000); \ldots; Emp \rightarrow scan\_index(a_1, salary, 30.000); \\
& t_1.a_1 \rightarrow raiseSalary(t_1.a_2); \ldots; t_n.a_1 \rightarrow raiseSalary(t_n.a_2); opened; s_1(t_1); \ldots; s_n(t_n) >
\end{aligned}
$$

$$
\begin{aligned}
T' \;=\; & TRACE(P') \;= \\
& < Emp \rightarrow open\_scan\_index(a_1, salary, 30.000); opened; \\
& Emp \rightarrow scan\_index(a_1, salary, 30.000); t_1.a_1 \rightarrow raiseSalary(t_1.a_2); s_1(t_1); \ldots; \\
& Emp \rightarrow scan\_index(a_1, salary, 30.000); t_n.a_1 \rightarrow raiseSalary(t_n.a_2); s_n(t_n); \\
& Emp \rightarrow scan\_index(a_1, salary, 30.000) >
\end{aligned}
$$

where $t_i = [a_1 : oi, a_2 : 10]$ and $s_i = [a_1 : oi, a_2 : 10, a_3 : b_i]$, $o_i \in extension(Emp)$, $b_i \in \{true, false\}$ for $i = 1, \ldots, n$

By comparing T and T' we see that removing the $collect$ operator changes the execution order of invocations of $raiseSalary$ and $scan\_index$. These methods are in conflict if the latter is invoked for the class Emp and the property $salary$. Thus, P' is not an admissible or valid EP since it cannot be executed deterministically.

It follows that if all methods which occur in the trace commute, the evaluation of a record-oriented EP P' leads to the same result and terminal database state as the evaluation of its corresponding set-oriented EP P, and P' is then a valid EP. However, commutativity between all methods is not necessary in order to guarantee a deterministic evaluation. This will be investigated more closely in the following.

4.2 Identifying Admissible Record-Oriented and Mixed EPs

We implicitly referred to commutativity as total commutativity when we observed that an EP is executed deterministically if all methods which occur in its trace commute. However, total commutativity between all methods is not necessary, as we will show with the following example.

Example 5 The following two EPs can be formulated for the declarative update in Example 1:

$$
\begin{aligned}
P \;=\; & map < a_5, raiseSalary, a_1, < a_4 >> (\\
& map\_const < a_4, 10 > (\\
& \quad select < a_2, >, a_3 > (\\
& \quad\quad map\_const < a_3, 30.000 > (\\
& \quad\quad\quad collect(map < a_2, salary, a_1 > (\\
& \quad\quad\quad\quad collect(get < a_1, Emp >)))))))
\end{aligned}
$$

$$
\begin{aligned}
P' \;=\; & map < a_5, raiseSalary, a_1, < a_4 >> (\\
& map\_const < a_4, 10 > (\\
& \quad select < a_2, >, a_3 > (\\
& \quad\quad map\_const < a_3, 30.000 > (\\
& \quad\quad\quad map < a_2, salary, a_1 > (\\
& \quad\quad\quad\quad collect(get < a_1, Emp >))))))
\end{aligned}
$$

P' is a valid alternative EP if the traces T and T' of P and P', respectively, can be transformed into each other by exchanging commuting method invocations:

$$
\begin{aligned}
T \;&=\; TRACE(P)\\
&= < Emp \rightarrow open\_scan(a_1);\; Emp \rightarrow scan(a_1); \ldots ; Emp \rightarrow scan(a_1);\\
&\quad x_1.a_1 \rightarrow salary(); \ldots ; x_n.a_1 \rightarrow salary(); opened;\\
&\quad t_1.a_1 \rightarrow raiseSalary(t_1.a_4)?; s_1(t_1)?; \ldots ; t_n.a_1 \rightarrow raiseSalary(t_n.a_4)?; s_n(t_n)? >\\
T' \;&=\; TRACE(P')\\
&= < Emp \rightarrow open\_scan(a_1); Emp \rightarrow scan(a_1); \ldots ; Emp \rightarrow scan(a_1); opened;\\
&\quad x_1.a_1 \rightarrow salary(); t_1.a_1 \rightarrow raiseSalary(t_1.a_4)?; s_1(t_1)?;\\
&\quad \ldots ; x_n.a_1 \rightarrow salary(); t_n.a_1 \rightarrow raiseSalary(t_n.a_4)?; s_n(t_n)? >
\end{aligned}
$$

where $x_i = [a_1 : o_i]$ and t_i, s_i are defined analogously as in Example 4 for $i = 1, \ldots, n$.

The methods $salary$ and $raiseSalary$ in T are executed for the same receiver. They are in conflict, since the former reads the property $salary$ of an employee, while the latter changes this property. Thus T' cannot be derived from T by commuting non-conflicting method invocations. However, $salary$ and $raiseSalary$ do not commute totally, but partially, because changing the salary of an employee e does not have an influence on reading the salary of another employee e'. Since each employee's salary is read and written only once in this example, conflicts will actually not occur. P' can be evaluated deterministically, although not all invoked methods commute totally.

This example shows that partial commutativity can be sufficient. Nevertheless we have to make sure that the methods in question are executed only once for the same receive. This is the case if the record column which holds the identifiers of the receiver objects contains only unique values in the evaluation of P. We say that a component a is *unique* in the evaluation of an EP P, if the execution of P yields the set of records S with $a \in Ref(S)$ and $\forall s_1, s_2 \in S, s_1 \neq s_2 : s_1.a \neq s_2.a$

Obviously a is unique in the evaluation of P, if it has been generated by the operators get and $select\_index$, and if P does not contain a join operator or the operator $flat$. The former condition guarantees that a is created with unique values, while the latter ensures that uniqueness is maintained since records are not duplicated during subsequent processing. If these conditions are not fulfilled, we cannot decide whether a is unique or not, unless we have further information about the result of the evaluation of operators.

We have seen that in some cases partial commutativity is sufficient to ensure a deterministic evalua tion. However, there exist other types of conflicts that do not prevent a deterministic evaluation, as the following example illustrates.

Example 6 Assume that the class Emp has an additional property $department$ which holds the OID of an instance of a class $Department$. This class provides a property $floor$ which indicates where a department is located. We want to increase the salary of all employees which work in a department on the third floor and earn more than $30.000. For this declarative update, the following EP P can be given:

$$
\begin{aligned}
P \;=\; & map < a_9, raiseSalary, a_1, < a_8 >> (\\
& \quad map\_const < a_8, 10 > (\\
& \quad\quad nested\_loop\_join < a_5, >, a_4 > (\\
& \quad\quad\quad select < a_6, ==, a_7 > (\\
& \quad\quad\quad\quad map\_const < a_7, 3 > (\\
& \quad\quad\quad\quad\quad map < a_6, floor, a_5 > (\\
& \quad\quad\quad\quad\quad\quad get < a_5, Department >))),\\
& \quad\quad\quad map < a_4, department, a_1 > (\\
& \quad\quad\quad\quad select < a_2, >, a_3 > (\\
& \quad\quad\quad\quad\quad map\_const < a_3, 30.000 > (\\
& \quad\quad\quad\quad\quad\quad map < a_2, salary, a_1 > (\\
& \quad\quad\quad\quad\quad\quad\quad get < a_1, Emp >))))))
\end{aligned}
$$

$TRACE(P)$ contains invocations of the methods $raiseSalary$ and $salary$ which partially commute, but the component a_1 which holds the receiver objects for these operations may not be unique in the evaluation of P due to the join

in the query. However, if we take a look at the algorithm given for $nested\_loop\_join$ in Section 3.2, we see that the inner input - in analogy to outer and inner relation of a join - , i.e., P_2, is completely evaluated before the first pair of matching records is actually computed. Thus all method invocations in the evaluation of P_2 are executed before the update method is first invoked. An invocation of the update method cannot have an influence on the execution of any of the methods that occur in P_2, and conflicts between the update method and a method in the evaluation of P_2 can be ignored.

Since invocations of $salary$ occur in the evaluation of the inner input of the nested loop join, the conflict between $raiseSalary$ and $salary$ can be ignored, and thus P can be evaluated deterministically.

This observation also holds for the operators $diff$, and, if the operator $sort$ is used for sorting the inputs, for $merge\_join$. For $diff$, the inner input has to be computed completely before the first result record can be computed. For $merge\_join$, both inputs have to be sorted, and none of the inputs is computed completely before the first pair of matching records is computed. If sorting is performed explicitly using the operator $sort$, we have to distinguish whether the conflicting method is executed before or after the sorting. In the former case, a conflict can be ignored, since due to the set-oriented evaluation of $sort$ all invocations of the conflicting method are executed before the update method is first executed. In the latter case, the methods have to commute totally, because, due to the join, the uniqueness of columns in the resulting records cannot be guaranteed.

We generalize the observations of Example 5 and 6 in the following two theorems. The first theorem describes the conditions under which a stepwise transformation of a set-oriented to a record-oriented EP is possible:

Theorem 1 Let P and P' be EPs of the form

$$P = map < a, m, a_{rec}, < a_{p_1}, \ldots, a_{p_l} >> (op_1(\ldots(op_k(collect(E)\ldots)$$
$$P' = map < a, m, a_{rec}, < a_{p_1}, \ldots, a_{p_l} >> (op_1(\ldots(op_k(E)..)\ldots)$$

where $op_1 \ldots op_k$ are physical operators, but none of these operators is a $collect$ operator. If P is executed deterministically, then also P' is executed deterministically and yields the same result, i.e., the same sequence of records is generated and the same database state is reached, if one of the follow ing conditions is satisfied:

(i) E is one of the subplans $select(collect(Q)), map\_const(collect(Q)), nested\_loop\_join(collect(Q), R)$, $hash\_join(collect(Q), R), merge\_join(collect(Q), collect(R)), union(collect(Q), R), diff(collect(Q), R)$ where Q and R are arbitrary subplans, or

(ii) E is $map < a', m', a_{rec}, < a'_{p_1}, \ldots, a'_{p_{l'}} >> (collect(Q))$ where Q is an arbitrary subplan, a_{rec} is a unique component in the evaluation of Q, m and m' commute partially, and each $op_i, i = 1, \ldots, k-1$, is one of the operators *map*, *map_const*, or $select$, or

(iii) E is $map < a', m', a'_r ec, < a'_{p_1}, \ldots, a'_{p_{l'}} >> (collect(Q))$, where Q is an arbitrary subplan, and m and m' commute totally, or

(iv) E is $get < a', C >$ or $select\_index < a', C, p, v >$ and m and $C \rightarrow scan(a')$, and m and $C \rightarrow scan\_index(a', p, v)$, respectively, commute totally.

Proof:
The removal of a $collect$ operator permutes the sequence of method invocations in the trace. We have to show that this permutation does not change the invocation order of conflicting methods.

Case (i):
For these operators, removing the $collect$ operator has no impact on the trace.

Case (ii):
Removing the $collect$ operator leads to two kinds of exchanges. Those between invocations of m' and method invocations induced by $op_1, \ldots, op_k$ are uncritical as only read-only methods are involved. The critical permutations take place between invocations of m' and m. Since we assume that the values of a_{rec} are unique in the evaluation of Q, i.e.,

for the same receiver object m' is always executed *before* m, partial commutativity between m and m' - as required in the theorem - is sufficient in this case.
Case (iii) and (iv):
In case of total commutativity, removing the $collect$ operator is always possible, since conflicts do not occur between methods whose invocations are interchanged.

Remark: $flat$ and join operators were excluded from the possible intermediate operators $op_1, \ldots, op_k$ as these operators may lead to duplication of the values of a_{rec} in several records. In this case total commutativity is required. However, we can include $flat$ and $join$ operators in case we have additional in formation, e.g., from database integrity constraints. These constraints may indicate that the execution of the methods which are invoked during the evaluation of these operators do not lead to a duplication. That is, in the case of $flat$ the resulting sets of the method call have at most size 1, while in the case of joins a record from the inner input does only match with a single record from the outer input, and vice versa.

The second theorem shows that also within the inner inputs of binary operators a stepwise transformation from the set-oriented to the record-oriented strategy is possible.

Theorem 2 Let P and P' be EPs with

$$P = map < a, m, a_{rec}, < a_{p_1}, \ldots, a_{p_l} >> (op_1(\ldots op_i(bop(Q, op_{i+1}(\ldots op_{k-1}(\\ collect(op_k(collect(S)) \ldots)$$

$$P' = map < a, m, a_{rec}, < ap1, \ldots, a_{p_l} >> (op_1(\ldots op_i(bop(Q, op_{i+1}(\ldots op_{k-1}(\\ op_k(collect(S))) \ldots)$$

where $op_1, \ldots, op_k$ are physical operators (including binary operators with a constant second argument), bop is the operator $nested\_loop\_join$ or $diff$, and Q and S are subplans of P. If P is executed deterministically, then also P' is executed deterministically and yields the same result, i.e., the same sequence of records is generated and the same database state is reached.

Proof:
The removal of the $collect$ operator only permutes the sequence of method invocations which are executed during $OPEN(P)$, i.e., the order of the method invocations which occur before the entry $opened$ in the trace is changed. Each of these method invocations is executed before the first pair of matching records in the binary operator is computed, and thus before the update method m is first invoked in the evaluation of P. Thus an invocation of m cannot have an influence on the results of the execution of these method invocations. If a conflict between m and a method induced by an operator $op_{i+1} \ldots op_k$ is detected, the conflict can be ignored.

These theorems show that total commutativity between all methods is not required in order to ensure a deterministic evaluation. Regarding Theorem 2, it directly follows that choosing the outer and inner input of the binary operators $nested\_loop\_join$ and $diff$ does not only have an influence on the evaluation cost and thus on the selection of the cheapest EP, but is also important for generating ad missible record-oriented and mixed EPs. E.g., if a conflict between a method which is invoked in the outer input of a nested loop join and the update method is detected - this would prevent a deterministic record-oriented evaluation - the conflict can be ignored if outer and inner input are exchanged.

5 Application of the Results

In our approach, the task of specifying conflicts explicitly, e.g., together with the database schema, might be rather expensive, since conflicts have to be specified between user-defined methods, and between user-defined methods on the one hand and system-defined methods on the other hand. However, conflicts can be derived automatically by exploiting traditional write/write and read/write conflicts, e.g., during the semantic analysis of the database schema at compile time. This is possible since we only utilize state-independent information for the specification of conflicts. The schema designer may then correct those cases where the methods semantically commute, although a conflict has been detected. A similar approach has already been suggested in [11].

5.1 Integration of the Conflict Test

The conflict test has to take place before the evaluation actually starts, such that its results can have an influence on the choice of the appropriate physical operators for the final EP. We shortly describe how the results of this paper will be integrated in the query optimizer of the object-oriented DBMS VODAK [17].

In VODAK we follow a rule-based approach for query optimization based on the Volcano optimizer generator [1][7]. The optimizer may enforce constraints in the generation of (sub)plans by using so-called *physical properties* (for example sortedness). According to Theorem 1 such constraints may occur when ommitting the $collect$ operator in a subexpression of an expression with a map operator on top. Theorem 2 identifies cases where the $collect$ operator can be savely omitted. In order to satisfy the constraints introduced by Theorem 1, we include information about methods that lead to potential conflicts, into the physical properties. By default, the implementation rules map the operator to an expression covered by $collect$. If we omit the $collect$ operator and potentially case (ii) - (iv) of Theorem 1 can occur, we include this information into the physical properties of the plan generated. Later, when a map operator tries to use such a physical expression as subplan, the conflict test can be executed by comparing required physical properties with the provided physical properties. Theorem 2 allows us to identify cases where we can abandon some of the constraints in the physical properties, namely whenever they occur in the second argument of a physical join or $diff$ operator. In this way the optimizer can choose record-oriented and mixed plans whenever they are safe alternatives. Thus we can take advantage of record-oriented processing strategies which can be more efficient than the corresponding set-oriented ones, although conflicts between the methods which are invoked during the evaluation exist.

5.2 Performance

Since parallelism cannot be exploited in traditional database systems, the execution of a record-oriented EP P' takes as long as the execution of the corresponding set-oriented EP P, provided the database buffer is large enough to hold all records which have to be processed. However, if the database buffer is too small, in contrast to the execution of P', the execution of P requires repeated replacement and reloading of records in the database buffer. We can estimate this effort for replacing and loading of records for a particular class of queries as follows. Assume that P and P' consist of n algorithms (no selections or joins) which have to be executed consecutively. Each algorithm has to process k records. The database buffer can store i records, where $i \ll k$. Loading a record into the buffer needs l units of time, and replacing a record needs r units of time. Then the total time for loading and replacing records during the execution of P and P', respectively, is given as follows.

- for P: $t = i * l + (k - i) * n * (r + l)$[3]
- for P': $t' = i * l + (k - i) * (r + l)$

Thus, we can achieve, if $i \ll k$, approximately a factor of n speed-up in the time spent for buffer management.

To substantiate these considerations we have compared the execution time of record-oriented and corresponding set-oriented EPs for queries in VODAK (for this particular experiment it did not matter whether we considered queries or updates). The queries were posed to a 180 MB protein database which was developed within the DOCKING-D project [8] at GMD-IPSI. The execution of queries and updates in VODAK is realized analogously as described in Section 3. In case the database buffer was large enough, the execution time of record-oriented and corresponding set-oriented EPs was, as predicted, nearly equal (difference ca. 1%). If the database buffer was too small, the execution time differed considerably: in the worst case, the execution of a record-oriented EP was 3.65 times faster than the execution of the corresponding set-oriented EP. In this case, 1069 of 1197 instances of a class were selected first, and 15 method calls were then applied to the selected objects consecutively. The database buffer could store 120 objects on average.

[3] This is actually the best case when all records in the buffer are processed before replacing and reloading of unprocessed records starts. In the worst case records are always processed in the same order, and thus for the last $(n - 1)$ algorithms $(k - i)$ unprocessed records in the buffer are replaced and reloaded afterwards. In this case the time for replacing and loading records is $t_{worst} = i * l + (k - i) * r + (k - i) * l + 2 * (k - i) * (n - 1) * r + 2 * (k - i) * (n - 1) * l$

6 Conclusion

In this paper we have investigated non-determinism in the evaluation of record-oriented EPs for declarative updates. In the examined cases non-determinism is induced by conflicts between update and read-only methods that are invoked within the algorithms which are executed for the evaluation. We have developed a framework which allows to identify admissible record-oriented and mixed EPs. It is shown that some of the detected conflicts can be relaxed and even completely ignored, while a deterministic evaluation can still be guaranteed. We have sketched a possible realization of our concept within a rule-based query optimization framework.

Generally speaking, we have introduced a new optimization potential with regard to the efficient evaluation of declarative updates. The contribution of our paper is twofold. First, and most important, we have shown that not all existing conflicts actually cause a non-deterministic evaluation. Second, we have illustrated that the generation of record-oriented and mixed EPs which are executed deterministically can be ensured if a conflict relation is available for the methods, or, in general, for the database access operations which are executed during the evaluation, and if appropriate conflict tests are performed before the actual execution starts, e.g., during the query optimization process.

The formal framework we have developed in this paper can also be applied to more complex declarative updates than investigated here, e.g., containing several update methods, or containing update methods in the selection conditions. Investigating these cases will be part of our future work.

References

[1] Aberer K, Fischer G. Semantic Query Optimization for Methods in Object-Oriented Database Systems. In: Proceedings of the 11th International Conference on Data Engineering. IEEE Computer Society Press, 1995, pp. 70-79.

[2] Andries M, Cabibbo L, Paredaens J, Van den Bussche J. Applying an update method to a set of receivers, In: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. ACM Press, New York, 1995, pp. 208-218..

[3] Bernstein P, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems, Chapter 5. Addison-Wesley Publishing Company, 1987.

[4] Cattell R.G.G (Ed.). Object Databases: The ODMG-93 Standard, Release 1.1. Morgan Kaufmann Publishers, San Francisco, 1994.

[5] Fischer G. Updates in Object-Oriented Database Systems Caused by Method Calls in Queries. In: Proceedings of the 3rd EDRG Workshop on Updates and Constraints Handling in Advanced Database Systems, 1992, pp. 64-72.

[6] Graefe G. Query Evaluation Techniques for Large Databases. ACM Computing Survey 1993; 2: 73-170.

[7] Graefe G, McKenna W.J. The Volcano Optimizer Generator: Extensibility and Efficient Search. In: Proceedings of the 9th International Conference on Data Engineering. IEEE Computer Society Press, 1993, pp. 209-218.

[8] Hemm K, Aberer K, Hendlich M. Constituing a Receptor-Ligand Database from Quality-Enriched Data. In: Proceedings of the International Conference on Intelligent Systems in Molecular Biology 95 (ISMB 95), 1995.

[9] International Standards Organization, Database Language SQL2 and SQL3, international committee document. ISO/IEC JTC1/SC21 WG3 DBL SEL-3b, April 1990.

[10] Jarke M, Koch J. Query Optimization in Database Systems. ACM Computing Survey 1984; 2: pp. 111-152..

[11] Laasch C, Scholl M. Deterministic Semantics of Set-Oriented Update Sequences. In: Proceedings of the 9th International Conference on Data Engineering. IEEE Computer Society Press, 1993, pp. 4-13.

[12] Muth P, Rakow T. C, Weikum G, Broessler P, Hasse C. Semantic Concurrency Control in Object-Oriented Database Systems. In: Proceedings of the 9th International Conference on Data Engineering. IEEE Computer Soceity Press, 1993, pp. 233-242.

[13] Rowe L, Stonebraker M. The POSTGRES Data Model. In: Proceedings of the 13th International Conference of Very Large Data Bases. Morgan Kaufmann Publishers, 1987, pp. 83-96.

[14] Selinger P, Astrahan M, Chamberlain D, Lorie R, Price T. Access Path Selection in a Relational Database Management System. In: Proceedings of the ACM SIGMOD Conference. ACM Press, New York, 1979, pp. 23-34.

[15] Stonebraker M. Operating System Support for Database Management. Communications of the ACM 1981; 7: 412-418.

[16] Stonebraker M, Wong E, Kreps P, Held G. The Design and Implementation of INGRES. ACM Transactions on Database Systems 1976; 3: 189-222.

[17] VODAK V4.0 User Manual, GMD Technical Report No. 910, Sankt Augustin, April 1995.

[18] Weihl W. Commutativity-Based Concurrency Control for Abstract Data Types. IEEE Transactions on Computers 1988; 12: 1488-150.

[19] Wilschut A, Apers P, Flokstra J. Parallel Query Execution in PRISMA/DB. In: Proceedings of the PRISMA Workshop on Parallel Database Systems. Springer Verlag, 1990, pp. 424-433 (Lecture Notes in Computer Science No.503).