Transaction Optimization in Rule Databases

E. Bertino, B. Catania, G. Guerrini, D. Montesi
Dipartimento di Scienze dell'Informazione, Università di Milano
Via Comelico, 39 20133 Milano, Italy
{bertino,catania,guerrini}@disi.unige.it

Abstract

This paper proposes an approach to transaction optimization in rule databases. It is based on a new technique to express updates in rule languages based on a non-immediate update execution. This technique is used to statically characterize some properties of the rules and transactions. Those properties are used at run-time to detect and remove redundant/complementary updates which are useless, hence increasing the efficiency of transaction execution.

1 Introduction

The use of rule-based languages in the context of data and knowledge bases has been a primary focus on research in the past decade in the area of deductive database systems and in the current decade in the area of active database systems. The motivation for using rule-based languages in database systems are twofold. On one side they are easy to learn and understand. They are also more user-friendly and higher level than traditional programming and database languages. On the other side they extend the properties of relational domain calculus into a rule-based language, adding to relational-based languages inferential and reactive features. Many advanced database systems [9, 10, 12, 14, 15, 16] are based on rule languages. Moreover, current extensions to relational DBMS and object-oriented DBMS also provide some rule capabilities. Indeed, rules represent a powerful and simple language to relate two situations: the cause and the effect. A rule can be seen as $\texttt{cause} \rightarrow \texttt{effect}$. This rule should be read as follow: if there is a cause, then do the effect. The causes and the effects can vary over a broad spectrum. In the database area, we are basically interested to query and update large amounts of information, hence the main causes and effects are *queries* and *updates* (where with updates we mean deletions, insertions and modifications). In the following table we provide a possible classification of rules w.r.t. which are the causes and the effects.

Cause	Effect
`query`	`update`
`update`	`query`
`update`	`update`
`query`	`query`

The first row denotes the pattern of active rules (i.e. rules with side-effects). The second one denotes a query depending from an action. The third row represents the most procedural type of rule. It denotes an action depending from another action (triggers are rules of this form). The last one is a classical deductive rule, that is, $\mathtt{B_1}, \ldots, \mathtt{B_n} \rightarrow \mathtt{H}$[1]. As it can be seen from the above table, updates are very often present in rules (either as causes or as effects, or both). However, the semantics of rules containing updates is difficult to define. Indeed, in most proposals of rule languages with updates, the semantics of rules is dependent on the evaluation order of the rules. Therefore, those languages are not fully declarative.

A recent proposal of a rule language [5] overcomes the above problem[2]. This language provides updates in rules by providing at the same time a declarative semantics. In this language a rule has the form

$$\mathtt{B_1}, \ldots, \mathtt{B_n}, \mathtt{INS_1}, \ldots, \mathtt{INS_k}, \mathtt{DEL_{k+1}}, \ldots, \mathtt{DEL_t} \rightarrow \mathtt{H}$$

In the above rule $\mathtt{INS_1}, \ldots, \mathtt{INS_k}$ and $\mathtt{DEL_{k+1}}, \ldots, \mathtt{DEL_t}$ represent conjunctions of insertions and deletions respectively. They are also in conjunction with deductive atoms $\mathtt{B_1}, \ldots, \mathtt{B_n}$. Such rules allow an execution model where insertions and deletions have a non-immediate semantics, that is, they are not executed as soon as they are evaluated. Rather the insertions and deletions are collected and executed, if they are consistent, only at the end of the evaluation of the query. The above type of rule is an integration of the second and last rows in the table. Indeed, such rules may contain updates in the bodies as well as queries; queries in the bodies are used to pass "parameters" to the updates. The heads of such rules only contain queries. Note that even if from a syntactical point of view, our rule language may look similar to other rule languages with updates in the bodies, like $\mathcal{LDL}$

[1] In the tradition of declarative rules (which we will follow) the above rule is represented as $\mathtt{H} \leftarrow \mathtt{B_1}, \ldots, \mathtt{B_n}$.

[2] A first version of the language has been implemented as part of a project at the University of Genova.

[14] and DLP [13], with respect to these languages our language has a fully declarative semantics which is independent from the evaluation order of atoms (both query and update atoms) in the rules.

The main concern of this paper is related to the efficient execution of this kind of rules. In particular, studying efficient methods for transaction execution in this context is an important issue. The semantics of our language makes it possible, among other things, to statically characterize some properties of rule sets and of transactions. Such characterizations, introduced in [4], give information about errors that may arise such as aborts, inconsistencies and failures. From these characterizations, related to a set of rules or to a transaction, it is possible to statically find information on useless updates, generated during the execution of a transaction and, therefore, to remove them. When errors are prevented, and useless updates are removed, transactions are executed more efficiently. The aim of this paper is to present the techniques used to perform transaction optimization in rule databases and the architecture of the optimizer. Note that in the previous literature, as far as we know, there are no satisfactory techniques for this kind of problem. In particular, a general framework to optimize transaction has been proposed in [2] for relational databases. In the context of rule-based languages the presence of not fully declarative semantics for updates has not allowed the development of such kind of optimization. The optimization we propose may be regarded as a "semantic update optimization" and in this sense it is in some way related to semantic query optimization [7].

The structure of the paper is the following. Section 2 introduces the rule language and sketches its semantics. Section 3, which contains the original contribution of this paper, describes the optimization architecture and discusses the several optimization phases. Due to space limitations, we introduce no formal notion, but we only discuss our techniques and illustrate them by an example. Section 4 presents some conclusions and outlines future work.

2 Rule language

A Datalog program [6] consists of a set of *base relations* (EDB) and a set *rules* (IDB). Many extensions to Datalog have been proposed to express updates (see [1] for a survey). In the following we summarize a new approach based on non-immediate update semantics.

Update-Datalog (U-Datalog) is a rule language which allows declarative specification of updates in program rules. The execution model of U-Datalog consists of two phases, the marking phase and the update phase. The first phase collects the updates found during the evaluation process, without, however, executing them. During the update phase they are executed altogether only if they are ground and consistent. If the set is not consistent, or if it contains non-ground updates, the transaction is *aborted* and no update in the set is performed. The notion of consistency is an important one, in that it prevents a set of updates containing both an insertion and a deletion of the same fact to be executed. By contrast in DLP and $\mathcal{LDL}$, updates are executed as soon as they are evaluated, that is, they are executed as side effect of the derivation process. We recall now some basic notions on U-Datalog. It is defined by means of an instance of constraint logic programming schema (CLP) [11] called $CLP(\mathcal{AD})$ [5]. In the following we will assume the reader familiar with logic programming [3] and with CLP.

Updates in U-Datalog are in rule bodies. In addition we consider also bindings in rule bodies which are defined by means of a set of equations (this is related to the fact that U-Datalog is an instance of CLP). Updates to base predicates are expressed as a set of special atoms prefixed by $+$ (insertions) or $-$ (deletions). The predicates can be either extensional or intensional. Our language allows only updates to extensional predicates.

Definition 2.1 *(Extensional database) An extensional database, or* state, *EDB is a (possibly empty) set of ground (i.e., without variables) facts.*

In the following we denote with $EDB_i, i = 1, \ldots, n$, the possible extensional databases.

Definition 2.2 *(Intensional database) The* intensional database *IDB is a set of rules of the form*

$H \leftarrow b_1, \ldots, b_k, u_1, \ldots, u_s, B_1, \ldots, B_t.$

where H is a deductive atom, $B_1, \ldots, B_t$ (as in Datalog) is the query part, $u_1, \ldots, u_s$ is the update part and $b_1, \ldots, b_k$ is the binding part. The update and query parts cannot be both empty.

The intuitive meaning of a rule is: "if $B_1, \ldots, B_t$ is true, the bindings $b_1, \ldots, b_k$ and the updates $u_1, \ldots, u_s$ are consistent, then H is true". Note that we do not consider update in rule heads. The notion of consistency is given informally. Intuitively, the bindings $X = bob, X = tom$ are not consistent, while the bindings $X = Y, Y = bob$ are consistent. Similarly, the updates $+p(X), -p(X)$, i.e. complementary updates, are not consistent. The updates $+p(Y), -p(X)$ could be consistent if the related bindings were for example $X = tom, Y = bob$. By contrast with the bindings $X = tom, Y = tom$, $+p(Y), -p(X)$ are not consistent.

Definition 2.3 *(Transaction) A* transaction *(or simple transaction) is a rule with no head of the form*

$b_1, \ldots, b_k, u_1, \ldots, u_s, B_1, \ldots, B_t$

where $B_i's, u_j's$ and $b_h's$ are as in Definition 2.2 and $B_1, \ldots, B_t$ cannot be empty.

Note that a transaction has a query component, i.e. it provides a set of bindings. The condition that $B_1, \ldots, B_t$ cannot be empty is due to the fact that the update phase must always follow the marking phase. Therefore, before updating a database, it must be queried in order to compute the bindings for the variables of the language. Following the tradition in the examples we prefix a transaction with the symbol '?'. A *complex transaction* T is a sequence of transaction $T_1; \ldots; T_n$.

Definition 2.4 *(U-Datalog) An* U-Datalog program *with updates (or database)* $DB = IDB \cup EDB$ *consists of the extensional database* EDB *and of the intensional database* IDB.

Example 2.1 *Consider* $EDB_i = q(b)$ *and*

$$\begin{aligned} IDB = \; & p(X) \leftarrow -q(X), q(X). \\ & r(X) \leftarrow +t(X), p(X). \\ & s(X) \leftarrow t(X). \end{aligned}$$

The transaction $T_1 = ?r(X)$ *evaluated in* $EDB_i \cup IDB$ *computes the binding* $X = b$ *and collects the updates* $-q(b), +t(b)$. *Informally the new extensional database* $EDB_{i+1} = t(b)$ *is the result of the application of these updates to* EDB_i. *The transaction* $T_2 = ?s(X)$ *evaluated in* $EDB_{i+1} \cup IDB$ *computes the binding* $X = b$ *and does not compute any update, thus the new extensional database is still* EDB_{i+1}. *The transaction* $T_3 = ? + q(X), s(X)$ *evaluated in* $EDB_{i+1} \cup IDB$ *computes the binding* $X = b$ *and collects the update* $+q(b)$, *thus the new extensional database is* $EDB_{i+2} = t(b), q(b)$. *The transaction* $T_4 = ? + q(X), p(X)$ *computes the binding* $X = b$, *and collects the updates* $+q(b), -q(b)$. *They are not consistent and therefore* T_4 *aborts. The transaction* $T_5 = ?X = a, -q(X), s(X)$ *fails, because* $t(a)$ *is not in* EDB_{i+2}, *and so no update is performed. The resulting EDB is* EDB_{i+2}. $\diamond$

The semantics of an U-Datalog program is given in four steps. The first step models the extensional and intensional components of the database as two separate components. Indeed due to the evolving nature of the EDB, the semantics of the database must be given in a compositional way, that is, in terms of the semantics of the IDB and EDB. The compositional semantics is based on the notion of open programs. An open program is a program in which information on a specified set of predicates is not defined. The intensional database of Example 2.1 is an open program. Indeed, the information concerning to the extensional predicate symbols are not defined. This step of the semantics is related to normal form for rules similarly to the notion of linear normal form introduced in [8]. We denote this normal form as $\mathcal{NF}(IDB)$. For non-recursive programs the normal form of a program looks very much like its unfolding, i.e. in each rule intensional atoms in the body are replaced by appropriate conjunctions of extensional atoms, thus obtaining rules with only extensional atoms in their body.

Example 2.2 *Consider IDB of the Example 2.1. The normal form is*

$$\begin{aligned} \mathcal{NF}(IDB) = \; & p(X) \leftarrow -q(X), q(X). \\ & r(X) \leftarrow +t(X), -q(X), q(X). \\ & s(X) \leftarrow t(X). \quad \diamond \end{aligned}$$

Due to space limitation we don't discuss here the normal form for recursive programs [4].

The second step semantics considers a database (extensional plus intensional parts) as a single component. Indeed, when querying a database (from a logical point of view) the distinction between extensional and intensional predicates is not relevant any more. This is the semantics of the marking phase. We note that database systems use as default a set-oriented semantics, that is, the query-answering process computes a set of answers. Therefore

$$Set(T, IDB \cup EDB) = \{\langle b_j, u_j \rangle \mid T \longmapsto^* \langle \tilde{b}_j, \tilde{u}_j \rangle\}$$

denotes the set of pairs (bindings and updates) computed as the consistent answers of the transaction T. Such answers can be computed in a top-down or bottom-up style. This semantics does not include the execution of the collected updates neither consider the transactional behavior. In order to model these features we define the semantics of a transaction T with respect to a database $IDB \cup EDB$. Before we define a function that performs the updates.

Definition 2.5 *Let* EDB_i *be the current database state and* u *is the consistent set of ground updates. Then the new database* EDB_{i+1} *is computed by means of the function* $\Delta : 2^{\mathcal{B}^e} \times 2^U \rightarrow 2^{\mathcal{B}^e}$ *as follows:*

$$\begin{aligned} \Delta(EDB_i, u) = \; & (EDB_i \setminus \{p(\tilde{t}) \mid -p(\tilde{t}) \in u\}) \\ & \cup \{p(\tilde{t'}) \mid +p(\tilde{t'}) \in u\} \end{aligned}$$

where $2^{\mathcal{B}^e}$ *is the set of possible database states and* 2^U *is the set of possible updates.*

The third step provides the transactional behavior, modeling the update phase; the hypothetical updates computed by the marking phase are executed with a transactional mechanism, i.e. an *all or nothing* style. As *observable property* of a transaction we consider *the set of answers, the database state* and *the result of the transaction* itself, which can be *Commit* o *Abort*. We consider the set of updates collected by the marking phase, to which the bindings have been applied. The set $\bar{u}$ is obtained as the union of all the updates gathered by the different solutions, appropriately instantiated. It can be a ground consistent set of updates. If so, the result of the marking phase is a set of bindings and a set of hypothetical updates. If the collected updates are consistent and ground, the new database state is computed and the transaction commits. If $\bar{u}$ is not ground or it is inconsistent, the transaction aborts. Note that in such a way we model a set-oriented transactional behavior. The set of all the possible observable properties is OSS. In the following we define the semantics of a transaction T with respect to the intensional database IDB as a function from extensional database to observable.

Definition 2.6 *(Semantics of a transaction) Let* $DB_i = IDB \cup EDB_i$ *be the database. The* semantics *of a transaction is denoted by the function* $\mathcal{S}_{IDB}(T) : 2^{\mathcal{B}^e} \longrightarrow OSS$. *If a transaction* T *has the form* $\tilde{b}, \tilde{u}, \tilde{G}$, *then*

$$\mathcal{S}_{IDB}(T)(EDB_i) = \begin{cases} Oss_{i+1} & \textit{if OK} \\ AbOss & \textit{otherwise} \end{cases}$$

where $Oss_{i+1} = \langle \{b_j \mid \langle b_j, u_j \rangle \in Set(T, DB_i)\},$

EDB_{i+1}, Commit$\rangle$, EDB_{i+1} is computed by means of $\Delta(EDB_i, \bar{u})$ and $AbOss = \langle \emptyset, EDB_i, Abort\rangle$. The condition OK expresses the fact that the set $\bar{u} = \bigcup_j u_j b_j$ is consistent, that is, there are no complementary ground updates. $u_j b_j$ denotes the ground updates obtained by substituting the variables in u_j with the ground terms associated with the variables in b_j.

The four step semantics is related to complex transactions i.e., sequences of transactions.

Definition 2.7 *(Sequence) Let $DB_i = IDB \cup EDB_i$ be the database and $T_1; T_2$ be a transaction. The semantics of $T_1; T_2$ is denoted by the function $\mathcal{S}_{IDB}(T_1; T_2) : OSS \longrightarrow OSS$.*

$$\mathcal{S}_{IDB}(T_1;T_2)(Oss_i) = \begin{cases} Oss_{i+2} & \text{if } OK \\ AbOss & \text{otherwise} \end{cases}$$

where $Oss_{i+2} = \mathcal{S}_{IDB}(T_2)(Oss_{i+1})$. $Oss_{i+1} = \mathcal{S}_{IDB}(T_1)(Oss_i)$ represents the observable of the database after the transaction T_1, $AbOss = \langle \emptyset, Oss_i.2, Abort\rangle$ and OK expresses the condition that $\mathcal{S}_{IDB}(T_2)(Oss_{i+1}).3 = Commit$ and $\mathcal{S}_{IDB}(T_1)(Oss_i).3 = Commit$.

Therefore, according to the above definition, the abort of a simple transaction in a sequence results in the abort of the entire sequence. This semantics induces an interesting equivalence between (complex) transactions. This is very important because we can transform a (complex) transaction into a semantically equivalent one which is computationally less expensive. Two transactions are semantically equivalent if they are observationally equivalent.

Example 2.3 *Consider the following intensional database IDB:*

$IDB = p(X) \leftarrow -q(X), q(X).$
$\quad r(X) \leftarrow +q(X), -q(Y), h(X), q(Y).$
$\quad s(X) \leftarrow t(X).$

Consider $EDB_i = \{q(b), q(a), h(b), t(c)\}$ and the transaction $?r(X); ?s(X)$. The transaction $?r(X)$ is evaluated in $EDB_i \cup IDB$. The marking phase returns the solution $< \{X = b, Y = a\}, \{+q(b), -q(a)\} >$. Note that the solution $< \{X = b\}, \{+q(b), -q(b)\} >$ is not returned by the marking phase because it is not consistent. The update phase executes all updates generated by the marking phase, because they are ground and consistent and therefore no abort condition arises. The following extensional database is computed:
$EDB_{i+1} = \Delta(EDB_i, \{+q(b), -q(a)\}) =$
$(EDB_i \setminus \{q(a)\}) \cup \{q(b)\} = \{q(b), h(b), t(c)\}$.
The execution of $?r(X)$ generates the observable property $< \{X = b\}, EDB_{i+1}, Commit >$. Now consider the transaction $?s(X)$, evaluated in $EDB_{i+1} \cup IDB$. The marking phase generates the solution $< \{X = c\}, \emptyset >$ where $\emptyset$ denotes the empty set. No update is generated and therefore no update is executed. The observable property returned by $?s(X)$, and by the transaction, is $< \{X = c\}, EDB_{i+1}, Commit >$. $\diamond$

3 Architecture of the transaction optimizer

In the framework of U-Datalog, which provides an execution model in which update execution is decoupled from the deduction process, optimization techniques can be developed whose goal is to reduce the update cost of transactions, i.e. to minimize the number of generated updates. Our transaction optimizer aims at transforming a transaction in an equivalent one, which is more efficient to execute with respect to the performed updates. Note that the transactional equivalence between the original transaction and the optimized one must be preserved, that is the execution of the optimized transaction must generate the same observable as the execution of the original transaction. For this reason we do not consider only state invariance, but also result and answer invariance for transactions. State invariance ensures that the database state obtained from the execution of the optimized transaction is the same state that would be obtained if the original transaction had been executed. Result invariance ensures that the original transaction and the optimized one have the same transactional behavior (commit/abort) while answer invariance ensures that the set of answers provided by the execution of the optimized transaction coincides with the one provided by the original transaction.

The overall scheme of the optimization process is shown in Figure 1. The optimization process can be divided into eight steps. Steps 1 to 4 are analysis steps. Steps 1 and 2 are a static analysis of the IDB, and are independent both from the database state and from the specific transaction. Therefore steps 1 and 2 are not executed for each transaction; rather they are only executed when the IDB is defined. In Step 1 the normal form is generated for the IDB.

In Step 2 from the normal form of the intensional database we obtain some properties related to the structure of the IDB, that hold for any EDB and transaction we consider. We refer to these properties as a *characterization* of the IDB, denoted as $\mathcal{C}(IDB)$. The characterization identifies which rules may cause inconsistency, abort or failure conditions during the marking phase of arbitrary transactions. Therefore, the characterization of an intensional database provides information about intensional predicates that, when invoked as part of the refutation process, may give rise to error conditions.

Steps 3 and 4 perform a static analysis of the transaction; therefore these steps are executed for each transaction which is submitted. Such analysis is first developed independently from the database state (step 3) and then refined keeping into account information on the EDB expressed in the form of state constraints. A state constraint is a property related to the contents of the database state. These properties lead to identify some relationships among extensional predicates.

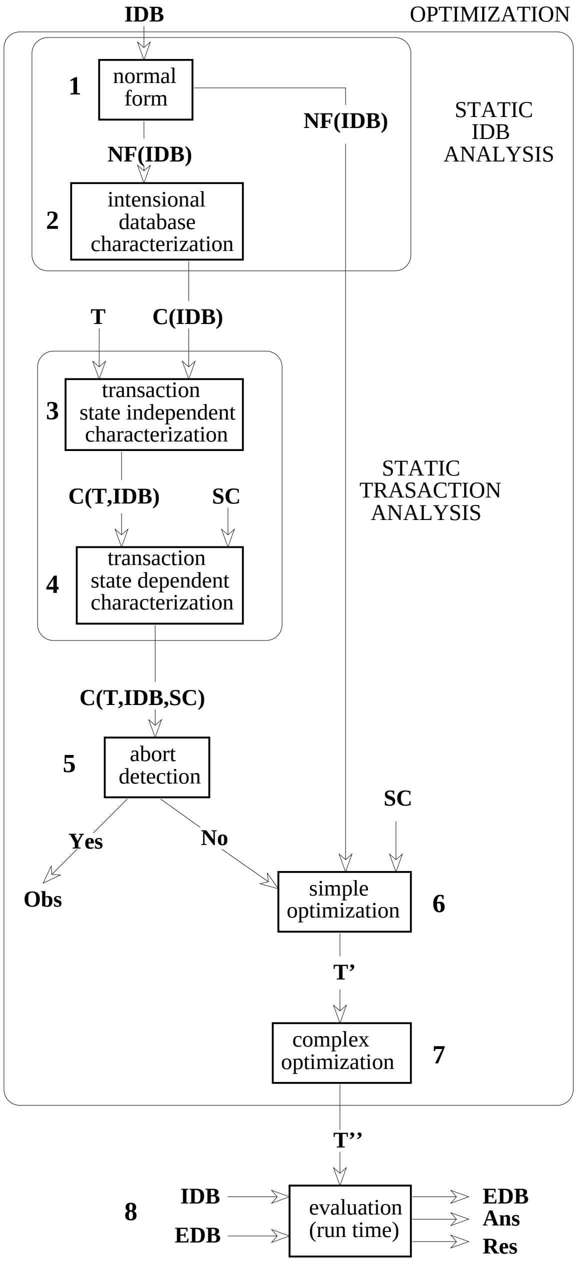

Figure 1: Steps in transaction optimization

The most important properties we are interested in characterizing are disjointness of predicates, inclusion relationship among predicates and emptiness of predicates. In our approach state constraints are formally represented as assertions among positive relational algebra expressions.

In Step 3 from the characterization of the intensional database the properties related to the transaction to be optimized are extracted. We specialize the characterization of the IDB to obtain more accurate information about the deduction of the considered transaction. The obtained characterization is called *state independent* characterization of the transaction because the obtained properties hold for any database state. In Step 4 the state independent characterization of the transaction is refined in a state dependent one by taking into account state constraints. The properties in the state dependent characterization are related to the deduction of the transaction on the considered IDB and an EDB satisfying the given set of constraints. The refined characterization is called *state dependent* one.

Steps 5 to 7 are the optimization steps, the first of which is related to abort detection, the second is related to optimization considering separately the simple transactions in the sequence (and so referred to as *simple optimization*) and finally the third is related to the optimization of the global sequence (referred to as *complex optimization*). Step 5 is concerned with the first optimization that can be performed on a transaction, i.e. the detection of aborts. Indeed, due to information provided by the characterization, we can statically determine whether the transaction aborts. If so, the observable that would be obtained from the execution of the transaction can be determined without actually executing the transaction. If an abort is detected the appropriate observable is returned to the user, without executing subsequent steps in the optimization.

Both complex and simple optimizations produce as output an annotated transaction. An annotated transaction is a sequence of simple annotated transactions, i.e. of simple transactions each one coupled with an annotation set (a set of rule identifiers and update identifiers). This set is used to specify that, in the marking phase for the simple transaction, the rules listed in the annotation set must not be used and, analogously, the updates listed in the annotation set must not be performed. In such a way each simple transaction that constitutes the sequence is to be evaluated in an IDB modified according to the annotation set. Annotations are used to discard invariant and redundant updates and to not consider rules that are known to generate no solution for the given transaction.

Simple optimization, performed as Step 6, has the goal of detecting invariant updates and failures. Invariant updates are those that, when applied to a database state, do not modify it. An update adding facts already in the database, or removing facts not in the database, is invariant. Failure detection is the process of determining which rules are useless in the deduction of a given goal, in that such rules do not produce any solution. With respect to invariant updates two invariance levels have been devised: *invariance with respect to a rule*, if the marking phase for the transaction by using such rule produces solutions that do not alter the database state, and *invariance with respect to an update*, if the execution of such update by the transaction does not alter the database state. Annotations produced as output reflect these different invariance levels. Moreover a simple transaction, which is not the last in the complex transaction (recall that, by the semantic definition, the answers depend on the last goal in the sequence), that is known not to abort and not to produce any database change (for example because it fails) can be eliminated from the transaction.

Step 7, i.e. complex optimization, has the goal of detecting redundant updates in the context of the sequence. Redundant updates are updates that are either executed twice in a sequence or discarded by complementary updates executed after them in the sequence. Complex optimization detects updates whose execution would not affect the resulting observable, and may introduce further annotation.

Conditions have been devised ensuring update invariance and determining redundant updates in the context of a sequence, and algorithms have been developed for generating the appropriate annotated transactions [4].

Step 8 is the evaluation process, in which the annotated transaction produced as output of the optimization is executed against the DB. This step, which is not discussed in this paper due to space limitations, simply executes each simple transaction in the sequence on the appropriate database state (the one resulting from the execution of the previous goals of the sequence) and the appropriate IDB (taking into account the annotation set for the considered simple transaction). The choice of the evaluation technique to adopt is not discussed in this paper.

3.1 An illustrative example

In this section we illustrate with an example the optimization strategy, shown in Figure 1. The EDB we consider contains the following extensional predicates: $customer/3$, $order/3$, $quantity/3$, $supplier/3$, $supplierCustomer/2$ and $productName/2$, storing the following information:

- $customer(Name, Address, Balance)$ defines for each customer the $Name$, the $Address$ and the $Balance$
- $order(Number, Date, Name)$ defines for each order $Number$, the $Name$ of the customer and the $Date$
- $quantity(Number, Product, Quantity)$ defines for each order $Number$, the related $Product$ and its $Quantity$
- $supplier(Name, Product, Price)$ defines for each supplier the $Name$, a $Product$ supplied by him and its $Price$
- $supplierCustomer(Name, Product)$ defines the $Name$ of a person who is both a customer and a supplier of $Product$
- $productName(Name, Product)$ defining the $Name$ of a person who is either a customer ordering $Product$ or a supplier supplying $Product$.

The IDB we consider consists of the following intensional predicates. The number near the rules and the update atoms are used to identify them in the following.

Intensional Predicate 1 The following rule determines all the suppliers of a given customer.

$1 : p_1\ (Customer, Supplier) \leftarrow$
$\quad supplier(Supplier, Product, Price),$
$\quad order(Number, Date, Customer),$
$\quad quantity(Number, Product, Quantity)$

Intensional Predicate 2 The following rules update the extensional predicate $productName$ by inserting the name and the product either ordered by a customer or supplied by a supplier.

$2 : p_2\ (Product) \leftarrow$
$\quad 1 : +productName(Name, Product),$
$\quad order(N, Date, Name),$
$\quad quantity(N, Product, Q)$

$3 : p_2(Product) \leftarrow$
$\quad 1 : +productName(Name, Product),$
$\quad supplier(Name, Product, Price)$

Intensional Predicate 3 The following rule retrieves the name of suppliers, supplying product a at price 100.

$4 : p_3\ (Name) \leftarrow Product = a, Price = 100,$
$\quad supplier(Name, Product, Price)$

Intensional Predicate 4 The following rule modifies the price at which a given supplier supplies a certain product. (Note that this rule contains two update atoms).

$5 : p_4\ (Supplier, Product, NewPrice) \leftarrow$
$\quad 1 : -supplier(Supplier, Product, OldPrice),$
$\quad 2 : +supplier(Supplier, Product, NewPrice),$
$\quad supplier(Supplier, Product, OldPrice)$

Intensional Predicate 5 The following rule updates the extensional predicate $supplierCustomer$, inserting the name of customers who are also suppliers, having the balance equal to 10000 and living in $Houston$.

$6 : p_5\ (Name) \leftarrow Address = Houston,$
$\quad Balance = 10000,$
$\quad 1 : +supplierClient(Name, Product),$
$\quad customer(Name, Address, Balance),$
$\quad supplier(Name, Product, Price)$

Intensional Predicate 6 The following rule removes from the extensional database the facts about the predicate *supplierCustomer* related to customers having the balance equal to 10000.

$7 : p_6 \leftarrow Balance = 10000,$
$\quad 1 : -supplierCustomer(Name, Product),$
$\quad customer(Name, Address, Balance),$
$\quad supplier(Name, Product, Price)$

Intensional Predicate 7 The following rules update the extensional predicate *supplierCustomer*, removing facts related to suppliers supplying at least a product at price 100 and inserting facts related to suppliers supplying at least a product at price 200.

$8 : p_7 \leftarrow Price = 100,$
$\quad 1 : -supplierCustomer(Name, Product),$
$\quad supplier(Name, Product, Price)$

$9 : p_7 \leftarrow Price = 200,$
$\quad 1 : +supplierCustomer(Name, Product),$
$\quad supplier(Name, Product, Price),$
$\quad customer(Name, Address, Balance)$

We now show how the various steps in the optimization (cf. Figure 1) are applied to the above IDB, by presenting two examples.

EXAMPLE 1

STEP 1

The first step generates the normal form of the intensional database. In this case, IDB does not contain any recursive clauses, and each rule has in its body only atoms on extensional predicates. Therefore, the normal form of IDB coincides with IDB, i.e. $\mathcal{NF}(IDB) = IDB$.

STEP 2

The second step generates a characterization of the normal form of IDB. We recall that a characterization of an intensional databases is a set of properties related to abort/commit, inconsistencies, success/failures conditions. These conditions can be obtained from $\mathcal{NF}(IDB)$, independently from the considered transaction. In the following, we illustrate the conditions generated for every single predicate of the example IDB.

Intensional Predicate 1 This predicate does not contain any update. Therefore, no abort can arise when this predicate is evaluated. This information is represented as: $\langle p_1, -, CM\rangle$, where CM denotes commit.

Intensional Predicate 2 The update parts of the rules defining this predicate do not contain complementary update atoms, i.e. $+p(\tilde{t})$, $-p(\tilde{t})$. Therefore, inconsistent solutions cannot arise. Moreover, every update atom shares its variable with a query atom. Therefore, non-ground updates cannot be generated by this predicate. This fact is represented as: $\langle p_2, -, CM\rangle$.

Intensional Predicate 3 This predicate generates the same conditions as those generated for predicates 1 and 2.

Intensional Predicate 4 The update part of the rule defining p_4 contains two complementary update atoms. These updates might generate some inconsistent solutions or some abort conditions. Moreover, the update atom 5.2 contains a variable not bound by a query or binding atom. This fact might lead to some non-ground updates, i.e. to some abort conditions. These facts can be represented as: $\langle p_4, (5.1, 5.2), PI\rangle$, $\langle p_4, (5.1, 5.2), PA\rangle$, $\langle p_4, 5.2, PA\rangle$, where PI denotes potential inconsistencies and PA denotes potential abort.

Intensional Predicates 5 and 6 These predicates generate only a commit condition, as that in predicate 1.

Intensional Predicate 7 This predicate is defined by two rules, containing complementary updates. In this case too, an abort condition for inconsistent updates can be generated. This condition is represented as: $\langle p_7, (8.1, 9.1), PA\rangle$.

Therefore at the end of Step 2 the obtained characterization $\mathcal{C}(IDB)$ is the following:

$\langle p_1, -, CM\rangle$, $\langle p_4, (5.1, 5.2), PI\rangle$, $\langle p_5, -, CM\rangle$,
$\langle p_2, -, CM\rangle$, $\langle p_4, (5.1, 5.2), PI\rangle$, $\langle p_6, -, CM\rangle$,
$\langle p_3, -, CM\rangle$, $\langle p_4, 5.2, PA\rangle$, $\langle p_7, (8.1, 9.1), PA\rangle$

where PF denotes potential failure. The above characterization is then stored by the system and used as the basis of optimization each time a transaction is submitted.

STEP 3

Suppose that the following transaction is submitted:
$T = G_1;\ G_2;\ G_3;\ G_4;\ G_5;\ G_6$ where
$G_1 = ?p_7$, $G_2 = ?p_2(Product)$,
$G_3 = ?p_5(Name)$, $G_4 = ?p_1(Customer, Supplier)$,
$G_5 = ?p_6$, $G_6 = ?p_3(Name)$

Upon submission of this transaction, the third step of the execution strategy generates a state independent characterization of the deduction of T in IDB. This characterization is obtained from the characterization of IDB, by considering the characterization properties related to predicates used during the execution of T. Moreover, we add a property for every rule for which $\mathcal{C}(IDB)$ does not contain a success or failure property. In this case the following state independent characterization $\mathcal{C}(T, IDB)$ is generated for T:

$\langle G_1, 8, PF\rangle$, $\langle G_1, 9, PF\rangle$, $\langle G_1, (8.1, 9.1), PA\rangle$,
$\langle G_2, -, CM\rangle$, $\langle G_2, 2, PF\rangle$, $\langle G_2, 3, PF\rangle$,
$\langle G_3, -, CM\rangle$, $\langle G_3, 6, PF\rangle$, $\langle G_4, -, CM\rangle$,
$\langle G_4, 1, PF\rangle$, $\langle G_5, -, CM\rangle$, $\langle G_5, 7, PF\rangle$,
$\langle G_6, -, CM\rangle$, $\langle G_6, 4, PF\rangle$

The above characterization states for example that the execution of goal G_4 will not generate any abort condition (i.e. G_4 will commit). However, rule r_1, used during the execution of G_4, may potentially fail (i.e. may not generate any solution). Note that the above characterization is generated without executing the transaction.

STEP 4
Now suppose that the following state constraint SC holds on the current EDB:

$\Pi_{1,2}(\sigma_{3=100}(SUPPLIER))\cap$
$\Pi_{1,2}(\sigma_{3=200}(SUPPLIER) \bowtie CUSTOMER) \neq \emptyset$

where Π represents the projection operator, σ represents the selection operator and $\bowtie$ represents the natural join operator. An uppercase name denotes the algebraic relation corresponding to an extensional predicate. This state constraint states that at least a supplier exists which supplies both a product at price 100 and a product at price 200. In this case, the potential abort condition $\langle G_1, (8.1, 9.1), PA\rangle$ becomes a certain abort condition $\langle G_1, -, AB\rangle$, In fact, because of the state constraint, goal G_1 will surely generate some inconsistent updates. The state dependent characterization $\mathcal{C}(T, IDB, SC)$ is the following:

$\langle G_1, 8, PF\rangle$, $\langle G_1, 9, PF\rangle$, $\langle G_1, -, AB\rangle$,
$\langle G_2, -, CM\rangle$, $\langle G_2, 2, PF\rangle$, $\langle G_2, 3, PF\rangle$,
$\langle G_3, -, CM\rangle$, $\langle G_3, 6, PF\rangle$, $\langle G_4, -, CM\rangle$,
$\langle G_4, 1, PF\rangle$, $\langle G_5, -, CM\rangle$, $\langle G_5, 7, PF\rangle$,
$\langle G_6, -, CM\rangle$, $\langle G_6, 4, PF\rangle$

Note that the state dependent characterization has the effect of determining for some potential error situations that they will arise for one given set of state constraints.

STEP 5
Because the characterization of T contains a certain abort condition, $\langle G_1, -, AB\rangle$, the outcome of T would be abort. Therefore, the transaction is not executed at all, and an abort result is returned to the user, together with an empty set of answers and the database state preceding the execution of the transaction.

EXAMPLE 2
This example illustrates the optimization of the transaction in Example 1 when different state constraints hold. Note that steps 1, 2 and 3 of the optimization are the same as those of Example 1, and thus we omit them.

STEP 4
Suppose that the following state constraints hold on the current EDB:

$ORDER = \emptyset$
$\Pi_{1,2}(SUPPLIER) \subset SUPPLIERCUSTOMER.$

In this case, rule 1 cannot generate any solutions during the execution of the goal G_4. Therefore the potential failure property $\langle G_4, 1, PF\rangle$ becomes a certain failure property, i.e. $\langle G_4, 1, FL\rangle$. The state dependent characterization of T is the following:

$\langle G_1, 8, PF\rangle$, $\langle G_1, 9, PF\rangle$, $\langle G_1, (8.1, 9.1), PA\rangle$,
$\langle G_2, -, CM\rangle$, $\langle G_2, 2, PF\rangle$, $\langle G_2, 3, PF\rangle$,
$\langle G_3, -, CM\rangle$, $\langle G_3, 6, PF\rangle$, $\langle G_4, -, CM\rangle$,
$\langle G_4, 1, FL\rangle$, $\langle G_5, -, CM\rangle$, $\langle G_5, 7, PF\rangle$,
$\langle G_6, -, CM\rangle$, $\langle G_6, 4, PF\rangle$

STEP 5
In this example, the characterization of T does not contain any certain abort conditions related to goals in T. Therefore, the abort detection step returns *no*.

STEP 6
From the state dependent characterization and from the set of state constraints, the simple optimization step recognizes two possible optimizations. First of all, as the goal G_4 uses only rule 1 during its execution and rule 1 does not generate any solution, this goal can be eliminated from T. Secondly the state constraint $\Pi_{1,2}(SUPPLIER) \subset SUPPLIERCUSTOMER$ allows to infer that the update atom 3.1 is invariant w.r.t. the extensional database. Therefore this update can be eliminated from the deduction of the goal G_2. We obtain the following annotated goal: $\langle G_2, U : 3.1\rangle$. The previous goal is executed as the goal G_2 on the intensional database obtained from IDB removing rule 3 and adding the rule

$p_2(Product) \leftarrow supplier(Name, Product, Price).$

The simple optimization returns the following transaction:

$T' = G_1; \langle G_2, U : 3.1\rangle; G_3; G_5; G_6.$

Note that the goal G_2 has been annotated by discarding the execution of the first update atom in rule 3. Moreover, goal G_4 has been removed from the transaction since it will not generate any solutions and thus will not modify the EDB state.

STEP 7
The last optimization step applies a complex optimization to the transaction obtained from the previous step. In this case, the optimizer recognizes that goal G_3 and goal G_5 generate some redundant updates. In fact G_3 generates updates on the predicate $supplierCustomer$ depending on the atoms

$Address = Houston,\ Balance = 10000,$
$customer(Name, Address, Balance),$
$supplier(Name, Product, Price),$

whereas the goal G_5 generates updates on the predicate $supplierCustomer$ depending on the atoms

$Balance = 10000,$
$customer(Name, Address, Balance),$
$supplier(Name, Product, Price).$

Therefore, the updates generated by G_3 are a subset of that generated by G_5. So, we can replace the goal G_3 by an annotation on the update atom identified by 6.1. We obtain the following optimized transaction:

$T'' = G_1; \langle G_2, U : 3.1\rangle; \langle G_3, U : 6.1\rangle; G_5; G_6.$

With respect to the original transaction T'' is more efficient in that: (i) it does not contain goal G_4, and (ii) updates atoms contained in rules 3 and 6 have been identified as redundant and thus will not be executed during the execution of T''.

STEP 8
The last step of the execution strategy evaluates T''

on the current database. We do not deal with this step due to space limitations. The only remark to be done is that the updates atoms contained in the annotations of the goals in the transaction (i.e. 3.1 for G_2 and 6.1 for G_3) will not be executed.

4 Conclusions

We have presented a rule-based language which provides updates within a declarative query language. Due to the non-immediate update execution interesting optimization on transactions can be performed. The main ideas of the optimization, based on static analysis of the database rules and transactions, are presented together with the architecture of the optimizer. An interesting problem which is still open and is currently under investigation is related to maintenance of state constraints. Indeed, state constraints can be either generated from the database state or updated due to database state changes depending on a cost model. A prototype of the optimizer is now under implementation.

References

[1] S. Abiteboul. Updates, a New Frontier. In M. Gyssens, J.Paredaens, and D. Van Gucht, editors, *Proc. Second Int'l Conf. on Database Theory*, volume 326 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, Berlin, 1988.

[2] S. Abiteboul and V. Vianu. Equivalence and Optimization of Relational Transactions. *Journal of the ACM*, 35(1):70–120, January 1988.

[3] K.R. Apt. Logic Programming. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–571. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990. Volume B: Formal Models and Semantics.

[4] E. Bertino, B. Catania, G. Guerrini, and D. Montesi. A Characterization of Intensional Databases in Constrained Datalog. Submitted for publication, 1993.

[5] E. Bertino, M. Martelli, and D. Montesi. Modeling Database Updates with Constraint Logic Programming. In U. W. Lipeck and B. Thalheim, editors, *Proc. Fourth Int'l Work. on Foundations of Models and Languages for Data and Objects*, pages 42–53, 1992.

[6] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

[7] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-Based Approach to Semantic Query Optimization. *ACM Transaction on Database Systems*, 15(2):162–207, June 1990.

[8] J. Han, K. Zeng, and T. Lu. Normalization of Linear Recursions in Deductive Databases. In *Proc. Nineth Int'l Conf. on Data Engineering*, pages 559–567. IEEE Computer Society Press, 1993.

[9] E. N. Hanson and J. Widom. Rule Processing in Active Database Systems. In L. Delcambre and F. Petry, editors, *Advances in Databases and Artificial Intelligence.* JAI Press, 1992. To appear.

[10] R. Hull and D. Jacobs. Language Constructs for Programming Active Databases. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 455–467, 1991.

[11] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, New York, USA, 1987.

[12] J. Kiernan, C. De Maindreville, and E. Simon. The Design and Implementation of an Extendible Deductive Database System. *SIGMOD Record*, 18(3):68–77, September 1989.

[13] S. Manchanda and D. S. Warren. A Logic-based Language for Database Updates. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, 1987.

[14] S. Naqvi and S. Tsur. *A Logic Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[15] R.Ramakrishan et al. The coral database system. In P. Buneman and S. Jajodia, editors, *Int'l Conf. ACM on Management of Data*, pages 544–546, 1993.

[16] J. Widom and S. J. Finkelstein. Set-Oriented Production Rule in Relational Databases Systems. In H. Garcia-Molina and H.V. Jagadish, editors, *Proc. Int'l Conf. ACM on Management of Data*, pages 259–270, 1990.