Eliminating Costly Redundant Computations from SQL Trigger Executions

François Llirbat, Françoise Fabret and Eric Simon

INRIA
78153 Le Chesnay,
France

email: first_name.last_name@inria.fr

Abstract

Active database systems are now in widespread use. The use of triggers in these systems, however, is difficult because of the complex interaction between triggers, transactions, and application programs. Repeated calculations of rules may incur costly redundant computations in rule conditions and actions. In this paper, we focus on active relational database systems supporting SQL triggers. In this context, we provide a powerful and complete solution to eliminate redundant computations of SQL triggers when they are costly. We define a model to describe programs, rules and their interactions. We provide algorithms to extract invariant subqueries from trigger's condition and action. We define heuristics to memorize the most "profitable" invariants. Finally, we develop a rewriting technique that enables to generate and execute the optimized code of SQL triggers.

1 Introduction

Active database systems (see e.g., [WC96]) enable the designer of the database schema to specify Event-Condition-Action rules, also called triggers. The database system automatically executes these rules within transactions, when it is appropriate, in accordance with the active rule execution model supported by the system. This provides a functionality that can be fruitfully exploited in many application domains, thereby explaining why most of the modern relational database systems provide trigger definition facilities.

In this paper, we focus on a specific optimization problem for active database applications, which is to avoid the redundant computation of costly subqueries arising in trigger's conditions or actions. Our framework is a relational active database system supporting SQL triggers that follow the syntax and semantics presented in [CPM96] and recently adopted for SQL3. To make the problem concrete, we consider the following example.

Example 1.1 We have a relation *Request* that contains requests for deliveries of packages (e.g., letter, parcel) to a given area, formulated at a given date with a given maximum delivery time (e.g., 12h, 24 h). Available private delivery services are provided in relation *Offer*. Each tuple indicates a service at a given date and a price for a given area. Relation *Contacts* lists all requests that can be satisfied by a given offer.

```
Request (userkey, date, area, delivtime, packtype)
Offer (userkey, area, date, price)
Contacts (demkey, offkey, date, price, area, packtype)
```

The for-each-row trigger depicted in Figure 1 is defined on relation *Request* and executed after an insertion command. Since it is a for-each-row trigger its action is executed for each inserted tuple. Its action first selects the best price among the offers that match the request's conditions. Then, it inserts into *Contacts* one tuple per offer that matches the request at the best price.

```
create trigger compute_offers_on_Request
after insert on Request
for each row
declare minprice integer;
 begin
  /* SQL statement S1 */
  select min(o.price) into minprice
  from Offer o
  where new.area = o.area and o.date > new.date
  and o.date  < new.date + new.delivtime;
  /* SQL statement S2*/
  insert into contacts (demkey, offkey, date, packtype,
                        price, area)
    select new.userkey, o.userkey, new.date,
           new.packtype, o.price, new.area
    from Offer o
    where o.price = minprice and new.area = o.area
    and o.date > new.date
    and o.date < new.date + new.delivtime;
 end
```

Figure 1: Trigger compute_offers_on_Request

Consider the program depicted in Figure 2. A user selects an area for deliveries and then for each delivery time, she selects a set of packtypes to be delivered. Then, the program performs a transaction that inserts the corresponding requests. After each insert into *Request* executed by the transaction, the above trigger is executed, that is its action is successively executed for every inserted tuple. Then the transaction terminates. Since, all inserted tuples have the same date, delivery time, and area, query Q1 in the trigger's action is exactly the same for every inserted tuple in the same transaction.

SIGMOD '97 AZ,USA

```
enter_Request (Userkey, Date)
 begin
  /* variable declaration */
  while ...
    get area into Area;
    while ...  /* get delivery times loop*/
      get delivery time into Deliv;
      get set of packtype into D;
      begin transaction /* enter_request transaction */
       /* SQL statement S0*/
       insert into Request (userkey, date, delivtime,
                            area, packtype)
        select(Userkey, Date, Deliv, Area, D.packtype)
        from D;
      end transaction
    end while
  end while
 end
```

Figure 2: Program enter_Request

A well known optimization technique can be applied to this example after noticing that the query Q1 is invariant during each transaction generated by **enter_Request**. If we memorize the result of Q1 into a table T1 after the first execution of the trigger's action corresponding to the first inserted tuple, we can reuse T1 in the subsequent executions of the trigger's action for the other inserted tuples. To implement this optimization, we would rewrite the trigger's action to account for T1. Of course, we implicitly assume that the rewrite will improve execution time.

However, this optimization requires careful attention for two reasons. First, one must observe that the rewritten version of the trigger is only valid when the trigger is executed by the *enter_Request* transaction. If another transaction performs an insertion into *Request*, without any knowledge on the values of the inserted tuples, then the original version of the trigger must be executed by that transaction. Thus, several versions of code may co-exist for a same trigger and we have to monitor their runtime execution depending on the transactional execution context.

Second, if the code for *enter_Request* is changed, one must check that the optimized version of the trigger associated with the execution of that procedure can still be used. That is, Q1 is still invariant during each execution of the transaction. Conversely, changing a program may enable new optimizations for existing triggers. A similar checking must be done if a new trigger is defined because it can interact with other existing triggers.

In the framework of a passive database application where all "business rules" are implemented within application programs, one can write as many procedures as needed to implement the rule of Figure 1 efficiently. Then, the writer of each transaction program simply selects an appropriate procedure and decides where to explicitly invoke it in her program. However, when a program changes, it is the programmer's responsibility to check that her decisions regarding the invocation of procedures are not compromised.

In active database applications, each trigger, whose event part is matched by an event instance generated by a transaction, is automatically executed as part of that transaction. Thus, if several alternative versions of a trigger are generated, an intermediate processing must take place between the application program and the database system to select a single appropriate version to execute.

The aim of this paper is to propose algorithmic solutions to the above problems. Our general goal is to generate an alternative version of a trigger when our knowledge about the execution context of that trigger enables us to eliminate costly redundant computations. Then, unlike passive applications, we want to select optimized versions of triggers at runtime transparently to application programmers. Finally, we want to automatically maintain optimized versions of triggers when the programs or the triggers are changed. It turns out that the techniques deployed in this paper can also be used to rewrite passive application programs (e.g., stored procedures).

More specifically, our main contributions are:

1. We model the interactions between transaction programs and triggers. Our model captures nested loops in programs, parameter passing in SQL statements and the flow of local variables assignments.

2. We provide algorithms to extract invariant subqueries from a trigger's condition or action with respect to a triggering path, that is a calling path starting from an SQL statement in a program and terminating at the trigger of interest.

3. We provide a simple and extensible heuristics to select profitable invariants from a set of possible invariants which capitalizes on existing query cost model and query rewriting rules.

4. We give an algorithm to monitor optimized versions of a given trigger according to the transactional context in which it executes, and show how to implement it on top of an existing active relational database system.

The paper is organized as follows. Section 2 outlines the rule execution semantics considered throughout the paper. Section 3 provides a model for transactions and rules. Section 4 shows how to extract repetitive invariant queries. Section 5 gives our heuristics to select the most profitable invariants to materialize. Section 6 shows how to monitor the optimized versions of a trigger at runtime. Section 7 describes related work, and Section 8 concludes and points out future directions of research.

2 Semantics of Rule Execution

Throughout this paper, we consider relational databases and Event-Condition-Action rules (henceforth, rules, or triggers) that consist of an event that causes the rule to be triggered, a condition that is checked when the rule is triggered, and an action that is executed when the rule is triggered and its condition is true. The triggering event is a data modification operation, i.e., an insertion, deletion or update, applied to a given relation. Thus, we only consider simple events. The condition is an SQL search condition over the database, and the action is an atomic procedure that may contain SQL statements combined with other procedural constructs.

In this paper, we illustrate our analysis and optimization techniques using the semantics for an SQL trigger language recently proposed in [CPM96]. We shall briefly characterize this semantics using the semantics parameters introduced in [FT95] and [WC96]. We refer the interested reader to [CPM96] for a more detailed presentation of the rule execution semantics.

Immediate C-A coupling mode: when a rule is triggered, its condition is first evaluated and then if it is true

the corresponding action is immediately executed within the same transaction than the transaction that triggered the rule.

Rule execution granularity: It indicates if the rule is instance-oriented (noted for-each-row granularity), or set-oriented (noted for-each-statement granularity). Both will be used in this paper. An instance-oriented rule is executed once for each instance of a database operation triggering the rule, whereas a set oriented-rule is executed once for all instances of a database operation triggering the rule. For instance, a rule whose event is an insert is triggered once for each tuple in the set of tuples inserted by an insert operation if it is instance-oriented. The same rule is triggered only once for the entire set of inserted tuples if it is set-oriented.

Rule processing granularity: It describes how often the points (henceforth, called *rule processing points*) occur at which rules may be processed. In this paper, we distinguish two kinds of rules: before rules are processed just before the triggering SQL statement while after rules are processed just after the triggering SQL statement.

Rule processing behavior: It specifies how the rules are executed at rule processing points. In particular, the operations of a rule action may trigger other rules. The semantics presented in [CPM96] adopts a recursive rule processing behavior: The execution of a rule recursively invokes the processing of the rules triggered by its action part. The recursive invocation is made at processing points within the action of the rules. If several rules are triggered at the same time, they are considered in a specific order specified by the designer or implicitly defined in the system (the creation order).

Transition relations: Each rule r triggered by the execution of an SQL statement s has access to the before values and after values of the tuples modified by s. These values are virtually stored in two *transition relations* usually called *Old* and *New* relations. If the rule execution granularity of r is for-each-statement, the entire transition relations are accessed by the condition part or the action part of r. If the rule execution granularity of r is for-each-row, one instance of r is executed for each modified tuple t in the transition relations, and the condition and action parts of r can only access the old and new values of the handled tuple t. Transition relations of a given SQL statement can be modified by before triggers that compute new values of the inserted or updated tuples, whereas after triggers never affect these relations.

3 Representation of Programs and Rules

We model transactions and rules as trees. The structure of the tree corresponds to the parse tree of the transaction or rule. We call these trees *abstract programs*. The abstract programs contain all the information essential to our solution.

3.1 Abstract Programs

The syntax tree for an abstract program P has a root node and one leaf node per variable assignment or SQL statement occurring in P. The control statements are represented by internal nodes in the usual way. The nodes of an abstract program correspond to the statements of the program. Thus, each node has a type, such as *while loop*, *SQL statement*, etc. For each node type, our analysis annotates the node with a description of information important to our solution. The set of node types are: (i) the *root* node, (ii) node types for SQL statements that insert, delete or replace tuples, (iii) node types for variable assignments, (iv) node types for program's control structures, and (v) nodes type for transactional SQL statements. To describe trees, we provide a simple syntax and then we associate some annotations with the three first types of nodes.

3.1.1 The Syntax of Abstract Programs

We restrict the programming control structures used in a program to sequential composition (noted by ';'), conditionals (noted `ifthen` and `else`), while loops (noted `whiledo`), and for loops (noted `for`). We intentionally omit representing the conditions in conditionals and while loops since they are not used in our analysis. A `for` statement has the following pattern: `for(` q,t`)` `do` *sequence* `od`, where q is a query, and t is a tuple variable iterating over the result of q. The transactional SQL statements we consider are begin of transaction and end of transaction (noted `bot` and `eot`). The syntax for SQL statements, variable assignment and the root node are simply identifiers that correspond to each statement.

Example 3.1 Consider Example 1.1: *enter_Request* is represented by the following abstract program $T0$:

```
whiledo sArea; whiledo sDeliv; sD; bot; S0; eot od od
```

`sArea`, `sDeliv`, `sD` are the identifiers for the assignment statements that respectively assign variables `Area`, `Deliv` and `D`. `S0` is the SQL statement that appears at the end of the procedure.

3.1.2 Annotation of a Root Node

The root node P is annotated by the pair $< V, P_a >$. Sets P_a and V denote respectively the set of input parameter variables and the set of local variables occurring anywhere in P. Thus, our program analysis gathers all references to variables and associates them with the root node. We distinguish between *relational*, *tuple*, and *domain* variables that respectively range over a relation instance, a tuple, and a domain of values. If v is a relational variable we denote the i^{th} column of v by $v.i$. If v a tuple variable, we denote the i^{th} component of v in the same way, by $v.i$.

Example 3.2 Let us take again Example 1.1. The root node `enter_Request` is annotated by: P_a = { `Userkey`, `Date`} and V= { `Area`, `Deliv`, `D` }.

3.1.3 Annotation of an SQL Statement

The annotation of an SQL statement s in a program $P =< V, P_a >$ is described by a tuple $< P_s, q_s, W_s, Old_s, New_s >$, where $P_s \subseteq (P_a \cup V)$ is the set of parameters of s. The query q_s is the query that appears in `insert` and `delete` SQL statements or the predicate of the `update` SQL statement. W_s is the set of write operations performed by s represented as follows. $+T$ (respectively $-T$) denotes an `insert` (respectively `delete`) operation into relation T. $[\pm T.i, F_i]$ denotes an update of column i in relation T and formula F_i specifies how column i is modified. Old_s and New_s are the queries that compute the transition relations associated with s. If s is a delete or an update statement, Old_s corresponds to the query q_s. If s is an insert, Old_s is undefined and New_s corresponds to the query q_s. Finally, if s is an update, then New_s

is given by the expression: *select* $(E_1, ..., E_k)$ *from* q_s, in which $E_i = F_i$, for all i such that $[\pm T.i, F_i]$ occurs in W_s.

Example 3.3 Consider again Example 1.1. Statement $S0$ of the *enter_Request* procedure is:

```
insert into Request (userkey, date, delivtime,
                     area, packtype)
select (Userkey, Date, Deliv, Area, D.packtype)
from D;
```

Its representation is $<P_{S0}, Q0, W_{S0}, Old_{S0}, New_{S0}>$, where $P_{S0} = \{$ `D, Userkey, Date, Deliv, Area`$\}$. $Q0 =$ `select(Userkey, Date, Deliv, Area, D.packtype) from D`. $W_{S0} = \{+$ `Request`$\}$. Old_{S0} is undefined and New_{S0} corresponds to $Q0$.

3.1.4 Annotation of a variable assignment

We annotate variable assignment statements depending on the type of assignment. $<x, P_s>$ annotates an assignment s of the form x:=*expr* where x is a domain variable, and P_s is a set of *domain* variables that appear in *expr*. To simplify the presentation, we restrict assignment statements of this form to contain *only* domain variables. Lifting this restriction is a straightforward extension of our work, that we do not consider further. $<x, q, P_s>$ annotates an assignment s of a relational or a domain variable x using a query q of the form "`select...into` x `from...where...`". $P_s \subseteq (V \cup P_a)$ is the set of parameters of q[1].

Example 3.4 Consider again Example 1.1 and take the abstract program of the *enter_Request* procedure in Example 3.1. We have the following assignments statements: $sArea = < Area, - >$, $sDeliv = < Deliv, - >$, $sD = < D, -, ->$[2].

3.2 Abstract Representation of Rules

We consider a rule r as an abstract program $<V, P_a>$ with a root node annotated with two additional descriptors, noted *Trig*, and *Change*. P_a only consists of two relational parameters noted New_r, and Old_r that contain the value of the transition relations accessed by r. V contains the local variables occurring in the action of r. *Trig* specifies the triggering operation of r. Descriptor *Change*, particularly suited for the before rules, indicates which columns of New_r are possibly modified by the rule. *Change* is a vector [e_1, ..., e_k] where k is the arity of New_r, and for i in {1, ..., k}, e_i = *true* if the ith column of New_r is modified and *false* otherwise. Recall that after rules do not change the transition tables. Thus, *Change* = [*false*, ..., *false*] if r is an after rule.

The structure of the abstract program for r encodes the rule execution granularity in an uniform way. If the granularity of the rule is for-each-statement, then r's abstract program is:

```
condition0; ifthen actionprogram0;
```

where `condition0` (if any) is the SQL assignment statement `C := select ... from ... where ...` computing the condition of r and `actionprogram0` is the abstract program associated with the action part of r.

If the granularity of the rule is for-each-row, r's abstract program is:

```
for(q,t) do condition1; ifthen actionprogram1 od;
```

where `q` is the query which reads the transition relations accessed by r. The domain of `t` is the result of `q`, namely it is either the instances of Old_r, New_r, or $Old_r \bowtie New_r$[3] The statement `condition1` tests if `actionprogram1` will be applied for the tuple `t`.

Example 3.5 The for-each-row rule of Example 1.1 is represented by the following abstract program:

```
T1 := for(Qnew,t) do S1; S2; od;
```

$V = \{t, minprice\}$, $P_a = \{New_{T1}\}$
$Change = \overrightarrow{false}$, $Trig = \{+Request\}$
$Qnew =$ `select * from` New_{T1},
$S1 = <minprice, Q1, \{t\}>$

```
Q1 = select min(o.price) from Offer o
     where t.area = o.area and o.date > t.date
     and o.date <t.date + t.delivtime.
```

$S2 = <\{t, minprice\}, Q2, \{+Contacts\}, Old_{S2}, New_{S2}>$

```
Q2 = select t.userkey, o.userkey, t.date,
               t.packtype, o.price, t.area)
     from Offer o
     where o.price = minprice and t.area = o.area
     and o.date > t.date and
     o.date < t.date + t.delivtime.
```

Old_{S2} is undefined
$New_{S2} = Q2$

4 Computing Invariant Queries

4.1 Preliminary definitions

We give definitions that are used throughout the section.

Definition 4.1 (*Database instances*). A database instance **I** consists of a set of relation instances, and a set of instances of variables or parameters. We shall use the following notations: **I**(R) denotes the instance of a relation schema R in **I**, q(**I**) denotes the instance of the output of a query q on **I** and **I**(v) denotes the instance of a tuple, domain or relational variable v in **I**.

Definition 4.2 (*Iterative regions and iterative trees*). Given a program $P =< V, P_a >$, we call *iterative region* of P any code fragment delimited by a *whiledo* or *for* statement. In the syntax tree for P, each subtree rooted at a for-node or at a whiledo-node denotes an iterative region of P. We model the iterative regions of a program P and the nesting between these regions by means of an *iterative tree* obtained from the syntax tree of P by discarding the conditionals (`ifthen`, `else`) and sequence nodes using the following transformation: If node n is a conditional or a sequence node then insert all its children nodes as children of its parent and delete n.

Example 4.2 Consider representations $T0$ and $T1$ of the *enter_Request* procedure and the for-each-row rule showed in Examples 3.1 and 3.5. The resulting iterative trees $T0$ and $T1$ are shown in Figure 3. Nodes $n0$ and $n1$ represent the iterative regions of $T0$ that are delimited by the `whiledo... od` statements and $n2$ represents the iterative region of $T1$ delimited by the `for(Qnew,t) do ... od` statement.

[1] Variable x is a domain variable if the query q computes an aggregate function.

[2] "$-$" means "unknown".

[3] when the rule is triggered by an update operation.

Definition 4.3 (*Triggers and Triggers⋆* Relationships). Let $\mathcal{P}$ be a set of abstract programs representing transactions and rules, $\mathcal{T}$ the set of corresponding iterative trees, and $\mathcal{S}$ the set of statements occurring in these programs. We describe the relationship between programs in $\mathcal{P}$ by means of a relation *Triggers* over $\mathcal{S} \times \mathcal{T}$. Given two elements (s,T) and (s',T') of $\mathcal{S} \times \mathcal{T}$, (s,T) *Triggers* (s',T') iff s is an SQL statement of T that triggers T' and s' is an SQL statement or an assignment of T'. We also define *Triggers⋆* as the extension of *Triggers* by transitivity.

Example 4.3 Consider the iterative trees in Figure 3. We have: (*S0*,*T0*) *Triggers* (*S1*,*T1*), and (*S0*,*T0*) *Triggers* (*S2*,*T1*). Now, suppose that we have another rule represented as *T3* in Figure 3, and such that (*S2*,*T1*) *Triggers* (*S3*,*T3*). Then, (*S0*,*T0*) *Triggers⋆* (*S3*,*T3*).

Definition 4.4 (*Triggering path*). Let (s,T) and (s',T') be two elements of $\mathcal{S} \times \mathcal{T}$ such that (s,T) *Triggers⋆* (s',T'). A *triggering path* from (s,T) to (s',T') is a calling path $\rho = (s_0,T_0) \ldots (s_p,T_p)$ of elements of $\mathcal{S} \times \mathcal{T}$ where $(s_0,T_0) = (s,T)$, $(s_p,T_p) = (s',T')$, and (s_i,T_i) *Triggers* (s_{i+1},T_{i+1}), for i in $\{0, \ldots, p-1\}$.

Example 4.4 In Figure 3, $\rho_1 = (S0,T0)\ (S1,T1)$ and $\rho_2 = (S0,T0)\ (S2,T1)$ are two triggering paths. Let us remark that a triggering path does not necessarily starts with a transaction program. For example, the triggering path $\rho_3 = (S2,T1)\ (S3,\ T3)$ begins with the rule program (i.e., $T1$).

Definition 4.5 (*Iterate set*). Let $\rho = (s_0,T_0) \ldots (s_p,T_p)$ be a triggering path, the *iterate set* of s_p in ρ, noted $Iterate_{s_p,\rho}$, is defined as follows. A node n of $\mathcal{S}$ is in $Iterate_{s_p,\rho}$ if:

1. there is no transactional node (**bot** or **eot**) in the subtree rooted at n and,
2. there exists $i \in \{0 \cdots p\}$ such that n is an ancestor of s_i in T_i.

Informally, $Iterate_{s_p,\rho}$ contains all the loop nodes that occur in ρ and whose execution may yield repetitive executions of s_p in the same transaction.

Example 4.5 Consider Figure 3 and the triggering paths of Example 4.4, we have: $Iterate_{S1,\rho_1} = Iterate_{S2,\rho_1} = \{\ n2\}$. $Iterate_{S3,\rho_3} = \{\ n2,\ n3\ \}$.

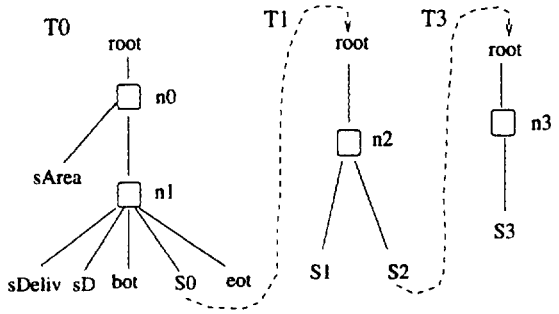

Figure 3: Iterative trees and triggering paths: a dotted arrow represents a Trigger relationship and a square box the root of an iterative region.

4.2 Invariant queries

Let us consider the triggering path ρ_1 in Figure 3 where $\rho_1 = (S0,T0)\ (S1,T1)$. When n_2 executes in ρ_1, $S1$, and therefore $Q1$, executes iteratively at specific points (called *execution points* of $S1$ wrt $(n_2,\ \rho_1)$), while the execution of $T0$ is suspended at $S0$. The number of such execution points is the total number of iterations of n_2. Our goal is to detect if the output of $Q1$ or of some subquery of $Q1$ is invariant during these repetitive executions. To infer this, we need to know if the parameters used in $Q1$ are also invariant during the execution of n_2 in ρ_1. We formalize this using *Invariance functions*.

Definition 4.6 (*Invariance function*). Let $\rho = (s_0,T_0)$, ..., (s_p,T_p) be a triggering path and n a node of $Iterate_{s_p,\rho}$. Let $\mathcal{I}_{n,\rho}$ be the set of database instances successively produced during the execution of n in ρ. We define the invariance function INV associated with (n,ρ) as follows:

1. If x is a domain variable, INV(x) = 1 if for any pair **I**, **I'** of instances in $\mathcal{I}_{n,\rho}$, $\mathbf{I}(x) = \mathbf{I'}(x)$; INV($x$) = 0 otherwise.
2. If x is a tuple variable of arity k, INV(x) returns a vector of arity k such that INV(x)[i] = 1 if for any pair **I**, **I'** of instances in $\mathcal{I}_{n,\rho}$, $\mathbf{I}(x.i) = \mathbf{I'}(x.i)$; INV($x$)[$i$] = 0 otherwise.
3. If x is a relational variable of arity k, INV(x) returns a vector of arity k that verifies the following property: if $e_1, \ldots, e_p$, $1 \leq e_i \leq k$, are all the indices for which INV(x)[i] = 1, then for any pair **I**, **I'** of instances in $\mathcal{I}_{n,\rho}$, $\Pi_{e_1,\ldots,e_p}(\mathbf{I}(x)) \stackrel{bag}{=} \Pi_{e_1,\ldots,e_p}(\mathbf{I'}(x))$[4]
4. If q is a query, INV(q) is defined similarly to item 3.

Informally, function INV takes a variable, a parameter or a query used in ρ and indicates if it is invariant during the execution of n in ρ.

Definition 4.7 (*Invariant query*). Let $\rho = (s_0,T_0)$, ..., (s_p,T_p) be a triggering path, n a node of $Iterate_{s_p,\rho}$, and q a query occurring in s_p, we say that query q is *invariant in n wrt ρ* if INV(q) = $\vec{1}$.

Example 4.6 Consider $\rho 1 = (S0,T0)\ (S1,T1)$. We give the INV function associated with ($n2$,$\rho 1$). For tuple variable t of $T1$, INV(t) = [1,1,1,1,0] because all the tuples inserted by statement $S0$ are equal on their **userkey**, **date**, **delivtime** and, **area** attributes. For query $Q1$, INV($Q1$) = [1]. Thus, $Q1$ is invariant in $n2$ wrt $\rho 1$.

4.3 The Invariance Computation algorithm

In $for(q,t)$ nodes, all the attribute values of variable t are taken from the output of q. Thus, if all tuples in the output of q have the same value on their i^{th} component, the i^{th} component of t is invariant within the for-node. This is formalized below.

Definition 4.8 (*Constant value function*). Let $\rho = (s_0,T_0)$ ... (s_p,T_p) be a triggering path and n a node of $Iterate_{s_p,\rho}$, we define the *constant value function* CONST associated with (n, ρ) as follows:

[4] $\stackrel{bag}{=}$ means equality between multi-sets

1. If q is a query, CONST(q) returns a vector with the same arity as the output of q and such that CONST(q)[i] = 1 if for any pair I, I' of instances in $\mathcal{I}_{n,\rho}$, all the tuples of $q(\mathbf{I}) \cup q(\mathbf{I'})$ are equal on their i^{th} component.

2. If x is a relational variable or parameter, CONST(x) is similarly defined.

To compute the INV and CONST functions associated with some node of an iterate set, we use the results of INV and CONST associated with previously computed nodes and propagate these results through the parameters that are passed to SQL queries, the dataflow of assignment statements, and the parameters passed between programs and rules via the transition relations.

We use three kinds of propagation rules: (i) the *query propagation* rules are used to compute CONST(q) and INV(q) for each query q occurring in ρ. (ii) the *variable propagation* rules are used to compute the INV and CONST functions on each local variable or parameter used in each T_i of ρ, and (iii) the *parameters propagation* rules are used to propagate the INV and CONST functions along ρ through the transition relations. Note that these propagation rules give sufficient conditions to determine the values of INV and CONST.

Query propagation rules: Given a triggering path $\rho = (s_0, T_0)...(s_p, T_p)$, and a node n of $iterate_{s_p,\rho}$, a query q occurring in an SQL statement s of T_i $(0 \leq i \leq p)$ is repeatedly executed during the execution of n in ρ iff n is in $iterate_{s,\rho}$. In this case, to determine if the query is invariant and/or constant valued we have to know :

- the database operations that may affect the base relation operands of q between two consecutive execution points of s. This set, noted $Write(n)$ corresponds to the set of write operations that are executed in the subtree rooted at n and in all the rules triggered by the statements in the subtree.
- the relational and domain parameters, and the components of tuple parameters of q that are invariant and constant valued between two consecutive execution points of s.

Definition 4.9 (*Write operations of a loop node*). Let $\rho = (s_0, T_0)...(s_p, T_p)$ be a triggering path and n a node of $Iterate_{s_p,\rho}$. Let T_j, $0 \leq j \leq p$, be the iterative tree that contains n, and $Stat(n)$ be the set of statements in the subtree of T_j rooted at n. The set of write operations generated by the execution of n is noted $Write(n)$ and defined as follows:

$$\begin{aligned} Write(n) \;=\; & \{op \in s.W_s \mid s \in Stat(n)\} \\ \cup\; & \{op \in s'.W_{s'}, \text{ for every } s' \text{ in some } T' \text{ s.t.} \\ & \exists s \in Stat(n)(s, T_j)\ Triggers^*\ (s', T'))\} \end{aligned}$$

We now present query propagation rules that allow to compute the INV and CONST functions for PSJ queries with aggregate functions. In the following we note $Param(q)$ the set of input parameters of q.

Base relation query propagation rule: If R is a base relation then

- CONST(R) = $\vec{0}$
- INV(R)[i] = 1 iff $Write(n)$ contains no operation of the form $+R$, $-R$, or $\pm R.i$.

Projection query propagation rules: If q = select $R.e_1, ..., R.e_k$ from R, where R is a relation of arity n, and $1 \leq e_i \leq n$, then

- CONST(q) is obtained from vector CONST(R) by only taking indices $e_1, ..., e_k$.
- INV(q) is similarly obtained from vector INV(R).

Selection query propagation rule: If q = select * from R where *predicate*, where R is a relation and *predicate* is in conjunctive normal form, then

- CONST(q)[i] = 1 iff :
  - CONST(R)[i] = 1, or
  - *predicate* has a conjunctive term of the form $R.i = \alpha$ where α is a constant, or
  - *predicate* has a conjunctive term of the form $R.i = Exp(x_1, ..., x_n)$ where each x_i is either a domain parameter of $Param(q)$ s.t. INV(x_k) = 1, or a component $t.j$ of a tuple parameter t of $Param(q)$ s.t. INV(t)[j] = 1.
- INV(q)[i] = 1 iff INV(R)[i] = 1 and for each domain parameter x of $Param(q)$, each component $t.j$ of t in $Param(q)$, and each $R.j$ occurring in *predicate*, INV(x) = 1, INV(t)[j] = 1 and INV(R)[j] = 1.

Cartesian product query propagation rule: If q = select * from $R1$, $R2$ then

- CONST(q) = CONST($R1$) $\diamond$ CONST($R2$)
- INV(q) = INV($R1$) $\diamond$ INV($R2$) if INV($R1$) $\neq \vec{0}$ and INV($R2$) $\neq \vec{0}$.

where $\diamond$ is a concatenation operator between vectors.

Special case: If q = select $R.1$, ..., $R.k$, x from R, where x is a domain variable or some component $t.i$ then

- CONST(q) = CONST(R) $\bullet$ INV(x)
- INV(q) = INV(R) $\bullet$ INV(x) if INV(R) $\neq \vec{0}$.

Join query propagation rule: We consider a join operation as a cartesian product followed by a selection.

Aggregate propagation rule: If q = select Agg from R, where Agg is an aggregate function then

- CONST(q) = $\vec{1}$ if INV(R) = $\vec{1}$.
- INV(q) = $\vec{1}$ if INV(R) = $\vec{1}$.

Example 4.7 Consider $\rho 1 = (S0, T0)\ (S1, T1)$ and assume that INV(t) = [1,1,1,1,0] for ($n2$,$\rho 1$). We first apply the selection propagation rule on the partial query $Q1'$ defined as:

```
select * from offer
where t.area = o.area and o.date > t.date
      and o.date < t.date + t.delivtime
```

We obtain INV($Q1'$) = [1,1,1,1] and CONST($Q1'$) = [0,1,0,0]. Then, we apply the projection propagation rule on query $Q1''$ = select price from $Q1'$. We obtain INV($Q1''$) = [1], and CONST($Q1''$) = [0]. Finally, by applying the aggregate propagation rule we get INV($Q1$) = CONST($Q1$) = [1].

Variable propagation rules: These rules take advantage of variable bindings inside a program. We distinguish three kinds of variable bindings: assignment statements of the form $<x, q, P>$ and $<x, P>$, and statements of the form $for(q, t)$. In the following rules, let T be an iterative tree or an iterative region in a path ρ.

Rule 1: If x is a domain variable in T then INV(x) = 1 if either :

- there is no assignment of x in T or,
- there is only one assignment s of x in T, which satisfies either (1) $s = <x, P>$ and foreach $y \in P$, INV(y) = 1, or (2) $s = <x, q, P>$ and CONST(q) = $\vec{1}$.

Rule 2: If x is a relational variable of T, then

- CONST(x)[i] = 1 if there is only one assignment statement $<x, q, P>$ of x in T and CONST(q)[i] = 1.
- INV(x)[i] = 1 if either:
  (1) there is no assignment of x in T or,
  (2) there is only one assignment statement $<x, q, P>$ of x in T and INV(q)[i] = 1.

Rule 3: Let t be a tuple variable occurring in a $for(q, t)$ statement of T then

- INV(t)[i] = 1 if CONST(q)[i] = 1.

Parameters propagation rules: Let $\rho = (s_0, T_0)...(s, T)(s', T')...(s_p, T_p)$ be a triggering path with $s = < P, q, s, Old_s, New_s>$. Let $Old_{T'}$ and $New_{T'}$ be the input transition relation parameters of T'.

Rule 4: For T', triggered by s in ρ:

- CONST($Old_{T'}$) = CONST(Old_s).
- INV($Old_{T'}$)= INV(Old_s).

Rule 5: Let $\mathcal{T}_s$ be the set of rules triggered by s other than T'.

- CONST($New_{T'}$)[j] = 1 if
  (1) CONST(New_s)[j] = 1 and
  (2) for any rule r in $\mathcal{T}_s$ $r.Change[j] = false$.
- INV($New_{T'}$)[j] = 1 if
  (1) INV(New_s)[j] = 1 and
  (2) for any rule r in $\mathcal{T}_s$ $r.Change[j] = false$.

Intuitively, Rules 4 and 5 propagate information about the columns of the transition relations produced by s, and Rule 5 accounts for the fact that before rules may change the values of New_s.

We are now able to present our *Invariant_Computation* algorithm. We shall use the following notations. Given a triggering path $\rho = (s_0, T_0)...(s_p, T_p)$ and a node n of T_i $(0 \leq i \leq p)$ such that $n \in Iterate_{s_p,\rho}$, $\mathcal{Q}$ denotes the set of queries occurring in ρ. $\mathcal{Q}^+ \subseteq \mathcal{Q}$ contains the queries occuring in all trees T_j of ρ s.t. $i < j \leq p$, and in the subtree rooted at n. $\mathcal{Q}^-$ = $\mathcal{Q}$ - $\mathcal{Q}^+$. $\mathcal{V}$ is the set of variables and parameters used in ρ. $\mathcal{V}^+ \subseteq \mathcal{V}$ contains the variables and parameters occuring in some T_j of ρ with $i < j \leq p$, plus the variables or parameters assigned in the subtree rooted at n in T_i. Finally, $\mathcal{V}^-$ = $\mathcal{V}$ - $\mathcal{V}^+$. The algorithm computes functions INV and CONST for all the variables, parameters and queries of $\mathcal{V}^+ \cup \mathcal{Q}^+$.

Invariance_Computation algorithm:
input: $\rho = (s_0, T_0)...(s_p, T_p)$ a triggering path, and n is an iterative node of T_i $(0 \leq i \leq p)$ such that $n \in Iterate_{s_p,\rho}$
output: the INV and CONST functions for $\mathcal{V}^+ \cup \mathcal{Q}^+$
Global knowledge: query, variables, parameters propagating rules
Initialization:
1: for each $q \in \mathcal{Q}^+$, INV(q) = $\vec{0}$
2: for each $q \in \mathcal{Q}^-$, INV(q) = $\vec{1}$
3: for each $x \in \mathcal{V}^-$ INV(x) = $\vec{1}$
4: for each $x \in \mathcal{V}^+$ INV(x) = $\vec{0}$.
5: for each relational variable x in $\mathcal{V}$ and each q in $\mathcal{Q}$,
CONST(x) = CONST(q) = $\vec{0}$
Propagation:
6: for j in {0, .., i} do /* Phase 1*/
repeat until function CONST does not change
Apply the query propagation rules to compute CONST for queries of T_j;
Apply rule 2 to compute CONST for the relational variables of T_j;
end repeat;
if $j \neq i$ then
Apply rules 4 and 5 to New_{s_j} and Old_{s_j};
od;
7: for j in {i,...,p} do /* Phase 2*/
repeat until functions CONST and INV does not change
Apply the variable propagation rules to the variables and parameters of $\mathcal{V}^+$ occuring in T_j;
Apply the query propagation rules to the queries of $\mathcal{Q}^+$ occurring in T_j;
end repeat
Apply rules 4 and 5 on New_{s_j} and Old_{s_j};
od;

Figure 4: Invariance_Computation algorithm

Example 4.8 Let us apply the algorithm to the triggering path $\rho 1 = (s_0, T_0),(s_1, T_1)$ and node $n2$. We have $\mathcal{Q}$ = $\mathcal{Q}^+$ = { $Q0$,New_{S0} = $Q0$, $Q1$, $Q2$ }, $\mathcal{V}$ = {*Area*, *Deliv*, *Date*, *Userkey*, D, t, *minprice*, New_{T1}}, $\mathcal{V}^+$ = {t, *minprice*, New_{T1}}, and $\mathcal{V}^-$ = {*Area*, *Deliv*, *Date*, *Userkey*, D}.

Initialization: By label 1, we obtain : INV($Q1$) =INV($Q2$) = $\vec{0}$, and INV($Q0$) = INV(New_{S0}) = $\vec{0}$. By label 3, we obtain: INV(*Area*) = INV(*Deliv*)= INV(*Date*) = INV(*Userkey*) = 1; INV(D) = [1]. By label 4, we have INV(*minprice*) = 0 and INV(New_{T1}) = INV(t) = [0,0,0,0,0]; By label 5, we obtain : CONST($Q1$) = CONST($Q2$) = $\vec{0}$ and, CONST($Q0$) = CONST(New_{S0}) = CONST(D) = CONST(New_{T1}) = $\vec{0}$

Label 6: Applying iteratively the cartesian product propagation rule to query $Q0$ we obtain CONST($Q0$) = [1,1,1,1,0] = CONST(New_{S0}). Then, we obtain CONST(New_{T1}) = [1,1,1,1,0] by rule 5.

Label 7: First $T = n2$; Then by applying rule 3, we obtain INV(t) = [1,1,1,1,0]. Applying the query propagation rules to $Q1$ (see Example 4.9) we obtain INV($Q1$) = CONST($Q1$) = [1]. Then applying rule 1 we obtain INV(*minprice*) = 1. Finally applying the query propagation rules to $Q2$ (selection, projection and cartesian product) we obtain INV($Q2$) = [1,1,1,0,1,1] and CONST($Q2$) = [1,0,1,0,1,1]. □

5 Profitable Query Subexpressions

5.1 Expression trees and DAGs

An *expression tree* for a query q is a binary tree, where each leaf node corresponds to an operand relation of q, and each non-leaf node contains an operator and has either one or

two children. Each non-leaf node is associated with the algebraic expression it computes and the algebraic expression computed by the root node is equivalent to q. Each internal node computes a sub-query of q. Generally, a query q has several expression trees, each of which describes a particular way of evaluating query q. Let us remark that two expressions trees for q also differ by the set of subexpressions they compute.

Expression DAGs [GM93] are used to compactly represent the space of equivalent expressions trees. We use them to represent the space of subexpressions of a query. An *expression DAG* for a query q is a bipartite directed acyclic graph with *equivalence* nodes and *operation* nodes. The root of the DAG is the equivalence node associated with q. Each node with no outgoing edge is an equivalence node called a "sink", which represents an operand relation of q. An equivalence node that has edges to one or more operation nodes is an internal equivalence node. An equivalence node M is labeled by the expression it computes, and the equivalence nodes in its subgraph represent subexpressions of M. An operation node contains an operator and has either one or two children that are equivalence nodes, and only one parent equivalence node.

Rule-based query optimizers [GM93] generate an expression DAG by applying a set of equivalence rules, starting from a given expression tree. Our heuristics is independent of the set of equivalence rules, and therefore describing them is out of the scope of this paper. We refer the interested reader to [jM93] for more details.

In the following, we assume that we have a cost model that enables (i) to estimate the size of any expression in the DAG, and (ii) to compute the cost of evaluating an expression M in the DAG from a set of subexpressions of M.

Example 5.1 Two expressions trees for the query $Q1$ of Example 1.1 and their resulting expression DAG are given in Figure 5. The Ei nodes represent the equivalence nodes of the DAG. The selection condition C is the where clause of query $Q1$. The selection condition C_1 is : `o.date < new.date + new.delivtime` and $C2$ contains the remaining predicates of C.

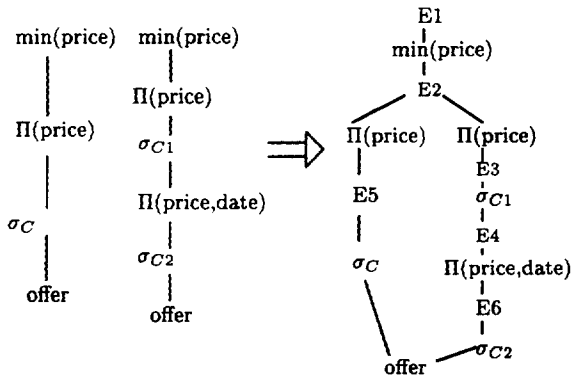

Figure 5: Expression trees and DAG for Q1

Definition 5.1 (*Annotated DAG*). Let $\rho = (s_0, T_0)\ldots (s_p, T_p)$ be a triggering path, q a query in s_p, the *annotated DAG* of q associated with ρ, noted $D(q,\rho)$, is derived from the expression DAG of q as follows: if M is an equivalence node of $D(q,\rho)$, q_M its associated expression, and n a node of $Iterate_{s_p,\rho}$ then n is in the annotation set of M iff q_M is invariant in n wrt ρ.

Example 5.2 Consider the iterative trees of Figure 6 that correspond to a slight variation of the iterative trees showed in Example 4.5, where node $n1$ is executed into a single transaction[5]. Let us take the triggering path $\rho_1 = (S_0.T_0')$ (S_1,T_1) and consider the query $Q1$ executed by $S1$. We have $Iterate_{(S_1,\rho_1)} = \{n2, n1\}$. As a consequence, $D(Q1,\rho_1)$ is annotated as follows: $E4$ and $E6$ are annotated with $\{n2, n1\}$, and the remaining equivalence nodes are annotated with $\{n2\}$.

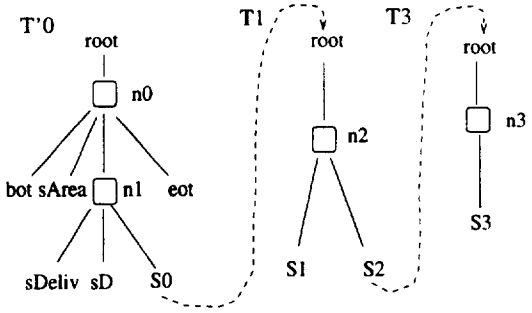

Figure 6: New iterative trees

5.2 Selecting a profitable set of queries

Definition 5.2 (*Profitability criterion*). Given a triggering path $\rho = (s_0, T_0)\ldots (s_p,T_p)$, a query q occurring in s_p, a node n in $Iterate_{s_p,\rho}$, and a sub-query q' of q, we say that q' is *profitable* for q in n wrt ρ if q' is invariant in n wrt ρ.

The rationale for that definition is that q' is a good candidate for caching because (i) it is a repetitive query, and (ii) since it is invariant, its caching does not incur any extra maintenance work. However, an annotated DAG $D(q,\rho)$ may have several annotated nodes (i.e., profitable expressions) among which some are useless to cache. For example, consider the annotated DAG of Example 5.2. Caching $E1$ and $E5$ is useless for two reasons: (i) $Q1$ can be computed using $E1$ or $E5$ only and, (ii) caching $E5$ to compute $E1$ is useless since $E5$ is invariant in the same iterative node as $E1$ $(n2)$.

Therefore, given a query q and a path ρ, there are several possible sets of profitable queries that are worthwhile to cache and the problem is to select the set that is expected to give the better performance improvement for ρ. We present an algorithm that selects a set of profitable expressions to cache with respect to a space threshold. It uses an elimination rule and is based on the notion of subdag, already introduced in [RSS96], which represents an expression tree for some equivalence node in a DAG.

Definition 5.3 (*Subdag*). Let M be an equivalence node in $D(Q,\rho)$, a subdag of M consists of a subgraph G of $D(Q,\rho)$ rooted at M such that:

1. each non sink equivalence node in G has exactly one child operation, and
2. all the child nodes of an operation node of G are in G

[5] We assume for example, that the user computes delivery times and packtypes from some base tables that may be updated by other transactions. In this case, all these computations are done within a same transaction.

Definition 5.4 (*Expressible query*). Let M be an equivalence node in $D(Q,\rho)$ and S a set of equivalence nodes reachable from M in $D(Q,\rho)$. M is *expressible* from S iff there exists a subgraph $G_{M,S}$ of $D(Q,\rho)$, embedded in a subdag of M and such that: (i) M is the root of $G_{M,S}$, and (ii) all the nodes of S are sinks in $G_{M,S}$.

Definition 5.5 (*Useful expression set*). Let M be an equivalence node in $D(Q,\rho)$ that is expressible from a set S of expressions. Then S is a *useful set* for M iff either M is not cached or S contains no annotated equivalence node N such that the annotation set of N is included in the annotation set of M.

Intuitively, this definition expresses that caching both N and M is useless as we illustrated in the previous example.

Definition 5.6 (*Cost function*). Let $D(Q,\rho)$ be an annotated DAG and M a non leaf node in $D(Q,\rho)$. Let S be the set of equivalence nodes reachable from M whose annotation set is non empty (i.e., each node of S is invariant in some loop node wrt ρ). If S_{cache} is the set of nodes in S that are cached, then the cost of M given S_{cache} is the cost of computing M using the cached queries of S_{cache}.

This simply expresses that there is no cost for maintaining the views of S_{cache}. However, there is a space occupancy cost associated with the caching of a set of queries. We assume that there is a fixed space limit for caching. Hence, we limit the total size of the cached queries along a triggering path. We also assume that the database relations represented by the sinks of the DAG are cached but their space occupancy is not counted in the total space occupied by the nodes of S_{cache}.

The algorithm given in Figure 7, takes as input an annotated DAG $D(Q,\rho)$ and a space limit Max_space. It computes a set S_{cache} and returns the DAG $D(Q,\rho)$ in which the nodes of S_{cache} are marked. The algorithm is iterative. It starts from the root node of $D(Q,\rho)$. At each iteration it selects the most profitable queries for computing the queries selected at the previous iteration, noted $\mathcal{M}$. It stops when there is no more equivalence node with a non empty annotated set, or when the space limit is exceeded.

This algorithm is naive, because the computation of S has a worst case complexity that is exponential in the number of annotated equivalence nodes of $D(Q,\rho)$. However, we can envision a greedy version of the algorithm [HRU96] by selecting at each iteration at most a fixed number of nodes whose caching produce a maximal benefit per unit of space for computing the nodes selected at the previous iterations.

Example 5.3 Consider the annotated DAG of Example 5.2 The *Select_invariant* algorithm applied to this DAG with an unbound Max_space works as follows:
At the first iteration, $\mathcal{M} = \{E1\}$ and S contains ($\{E1\}$), ($\{E2\}$), ($\{E3\}$), ($\{E4\}$), ($\{E5\}$), ($\{E6\}$), ($\{E3, E5\}$), ($\{E4, E5\}$) and, ($\{E6, E5\}$). Using the cost function, ($\{E1\}$) is selected for materialization.
At the second iteration, $\mathcal{M} = \{E1\}$ and S contains ($\{E4\}$) and ($\{E6\}$). ($\{E4\}$) is the more profitable subexpression and is selected for materialization.
At the third iteration, $\mathcal{M} = \{E4\}$ and S is an empty set since ($\{E6\}$ is not an useful set for $E4$.

Select_Invariant Algorithm:
Input: $D(Q,\rho)$ an annotated DAG and,
Max_space a space threshold
Output: S_{cache}, and $D(Q,\rho)$ where the nodes of S_{cache} are marked.
Global Knowledge: functions *Cost*, and *Size*
Initialization: $\mathcal{M} = \{Q\}$; $S_{cache} = \emptyset$;
Iteration:
repeat
Let $\mathcal{M} = \{M_1, \cdots M_k\}$ and,
Compute in S the set of tuples $(s_1, \cdots, s_k)$ such that:
1- For each i, s_i is an useful set for M_i and,
2- $size(s_1 \cup \cdots \cup s_k)$ is less than Max_space.
Choose $s = (s_1, \cdots, s_k)$ in S satisfying:
1- $cost(M_1, s_1) + \cdots + cost(M_k, s_k)$ is minimal in S and,
2- if several tuples satisfy item 1 then s is the set such that $size(s_1 \cup \cdots \cup s_k)$ is minimal
$\mathcal{M} = s_1 \cup \cdots \cup s_k$
$S_{cache} = S_{cache} \cup \mathcal{M}$
$Max\_space = Max\_space$ - $size(\mathcal{M})$
until there is no s to choose or $Max\_space \leq 0$
mark $D(Q,\rho)$ with S_{cache}.

Figure 7: Select_Invariant Algorithm

6 Monitoring optimized triggers

Using the algorithm of Section 5, we are able to select invariant expressions in a trigger with respect to a loop node in some triggering path, henceforth called *optimized triggering path*. Suppose that we use these invariants to generate an optimized version of the trigger with rewritten queries that account for the invariants. Then, we obtain one optimized version of the trigger per optimize triggering path. The purpose of this section is to show how to monitor optimized triggers on top of an existing active database system. It raises the following issues:

- How to determine in which triggering path we are at run time,
- How and when to store and compute the invariants, and
- How to select optimized versions of triggers at run time.

These techniques were tested on ORACLE V7.3 (see [LFS97] for more details).

6.1 Detection of triggering paths

Let $\mathcal{T} = \{T1, ..., Tn\}$ be a set of programs and rules. We associate with each SQL statement s in $\mathcal{T}$ a unique identifier *s_id*. This allows us to represent any triggering path of $\mathcal{T}$ by a sequence *s1_id*, ..., *sp_id* of identifiers. During the execution of programs and rules the current triggering path is maintained by means of a stack named *Pcall*. Before executing an SQL statement s, its identifier is pushed into the stack, and popped from the stack just after executing s. Thus, at each time of the execution, global variable *Pcall* contains the longest current path.

Pcall can be queried by means of the boolean function *Is_current* that takes a triggering path represented by a sequence of identifiers and that tests if *Pcall* contains this sequence.

Code rewriting rule R1: Given an optimized triggering path *s1_id*, ..., *sp_id*, then each statement *sj* ($1 \leq j \leq p$) is encapsulated by a block of the form:

```
begin push(Pcall, Sj_id ); Sj ; pop(Pcall, Sj_id ) end
```

6.2 Storage and computation of invariants

Invariants are managed as base relations [6]. To monitor the computation and the storage of invariants, we use a read-only relation INV, a main memory table INVDESC and the following rewriting rule:

Code rewriting rule R2: The code of each iterative region n involved in an optimized triggering path is encapsulated by the following block:

```
begin  INnode(n); code of n ; OUTnode(n) end
```

Relation INV(M_{id}, n) is used to store the marked annotated DAGs produced by the select_invariant algorithm. It associates each selected invariant M to its annotation set, that is, the set of iterative regions where M is invariant. The main memory table INVDESC($Mid, recompute, iter$) describes the current state of the invariants. Field *recompute* is a boolean, its value is true if the invariant must be recomputed and false otherwise. Field *iter* is an integer, it gives the current nesting level of the iterative regions where M is invariant. Table INVDESC is updated by means of three procedures *INnode*, *OUTnode* and *set_recompute_to_false*. *INnode* is called before entering an iterative region n. It selects from table INV the invariants in n and increments their *iter* attribute in table INVDESC. *OUTnode* is called at the end of an iterative region n. It decrements the *iter* attribute of the invariants in n and sets *recompute* to *true* if *iter* $= 0$. Procedure *set_recompute_to_false* is called after computing the new value of an invariant M. It sets the *recompute* attribute of M to *false*. The fields *recompute* and *iter* associated to an invariant M can respectively be obtained by using the *checkrecompute*(Mid) and *checkiter*(Mid) functions.

Generating the code to compute invariants: Let q be a query in T, ρ an optimized triggering path and $D(q, \rho)$ the marked annotated DAG of q wrt. ρ. The code generation algorithm visits $D(q, \rho)$ in a bottom up fashion. Each time it encounters a node M that is marked for materialization, the algorithm acts as follows:

1. It generates an identifier Mid.
2. It generates and executes the code of an SQL command that creates a relation of name M_Mid. This relation will be used to store the invariant.
3. It generates and executes the code of an SQL command that inserts into INV one tuple of the form (Mid, n) per annotation n occuring in the annotation set of M.
4. It generates and stores the code of a procedure called *ComputeM_Mid*.

Let M be a node in $D(q, \rho)$ that is marked for materialization, and $\mathcal{M} = \{m1, ..., mp\}$ the set of identifiers of the (marked) subexpressions of M. Then the code of *ComputeM_Mid* is generated using the following rule where *ComputeM_mi* ($1 \leq i \leq p$) is the procedure computing mi.

Code rewriting rule R3: The *ComputeM_Mid* parameters are the parameters of M. The *ComputeM_Mid* code is the following:

[6] A best solution is to store invariants into temporary relations provided that the underlying DBMS enables to reference temporary relations in SQL queries.

```
procedure  ComputeM_Mid(Param_Mid)  is
boolean :  validpath = true;
begin
 validpath := validpath AND  (checkiter(m1) > 0);
 if ((checkrecompute(m1) = true) AND (validpath))
    then ComputeM_m1(Param_m1);


 validpath := validpath AND  (checkiter(mp) > 0);
 if ((checkrecompute(mp) = true) AND (validpath))
    then ComputeM_mp(Param_mp);
 delete * from M_Mid;
 if (validpath)
    then insert into M_Mid  MS;
    else insert into M_Mid  MB;
 end if;
end;
```

MS is the expression that computes M using $\mathcal{M}$ and MB is the expression that computes M from the base relations. The boolean variable *validpath* is set to false if there is a subexpression mi of M such that *checkiter*(mi) $= 0$. Such situation arizes when there is no iterative region in the current triggering path where mi is invariant. In this case, M is computed using the base relations.

6.3 Selecting optimized query expressions

Let q be a query, and s the statement that uses the query q. Suppose that q has been optimized along a set of triggering paths $\mathcal{C} = \{c1, ..., cn\}$ ordered with respect to the inclusion relationship beetwen the paths (i.e. for any pair (cj, ci) of paths in $\mathcal{C}$, if cj is a subpath of ci then ci precedes cj in $\mathcal{C}$). Then the following rewriting rule is applied:

Code rewriting rule R4: s is replaced by the following code:

```
if     current_path = c1 then  Block_c1;
elsif  current_path = c2 then  Block_c2;


elsif  current_path = cn then  Block_cn;
else   s;
end
```

Each block Bloc_ci ($0 \leq i \leq n$) executes the optimized expression of q wrt. triggering path ci. The rationale for handling the paths in this ordering, is the fact that the code is optimized for various triggering paths with different lengths. For example, assume that q occurs in statement s of a rule r and that q is optimized wrt. the two following triggering paths $\rho = (s_0, P_0)(s_1, r_1)(s, r)$ and $\rho' = (s_1, r_1)(s, r)$. The optimization wrt. ρ' is more general since it is valid for any program that triggers r_1. On the other hand, the optimization of q wrt. ρ is more effective since new invariants may be detected due to the propagation of the context of P_0 along ρ. Ordering the triggering paths with respect to the inclusion relationship leads to systematically provide the more effective optimization.
Each block Bloc_ci contains the following code:

```
begin
 validpath := true;
 Compute_m1; ... Compute_mp;
 if (validpath) then si  else s;
end;
```

Statement si is obtained by replacing the initial expression of q in s by the optimized expression of q wrt. ci. The *Compute_mi* ($i \in \{1 \ldots p\}$) blocks contain the following code:

```
 validpath := validpath AND  (checkiter(mi) > 0);
if ((checkrecompute(mi) = true) AND (validpath)
  then  ComputeM_mi(Param_mi);
```

where the mi's ($i \in \{1, \cdots, p\}$) are the identifiers of the invariants used in the optimized expression of q in si.

6.4 Robustness

If one deletes a rule or a program T, it will make no effect on the performance of the remaining optimized code: Deleting T eliminates the occurrence of the triggering paths where T was involved but the optimization decisions of the remaining triggering paths are still valid and effective. Indeed, the structure of the programs involved in the remaining optimized triggering path are not changed and no new write operations are introduced.

Adding a rule r introduces new update operations in triggering paths. Such situation occurs when r is triggered by an SQL statement s of a program (or a rule) that is involved in an initially optimized triggering path ρ. Every iterative node n executing s in ρ may perform these new update operations and make its invariants no more valid. To deal with this problem, we first recompute annotation sets of all materialized invariants in ρ. Then we update table INV according to these new annotation sets. Recall that (i) code rewriting rule R2 guarantees that an invariant is no more computed when it is deleted from table INV [7] and, (ii) code rewriting rule R3 guarantees that queries are executed from base relations when they have invalid invariant subexpressions. This method ensures the correctness of the optimized code when rules are added. However, the efficiency of the optimization decreases when rules are added. At the very worst, all the initially optimized queries will execute without using invariants. A new optimization process is then necessary.

Adding a program introduces new write operations but these operations will never be triggered during the execution of an initially optimized triggering paths.

7 Related work

The optimization of terative computations optimization using cached expressions has been proposed in the context of programming languages [KP81] in the framework of the SETL language. The basic idea is to precompute and memorize subexpressions in auxiliary variables placed just before the loop and to use these results at each iteration in the loop. If the optimized loop is a part of another iterative region, the same method is recursively applied. These code optimization methods generally use heuristics to decide what subexpressions are *profitable* to save for a given iterative region.

The problem of optimizing repetitive evaluations of rules by caching intermediate results has been studied in the context of production rules. The proposed solutions are based on discrimination network for rules. *RETE* [For82] and *DBCond* [SLR93] materialize selection and join nodes while TREAT [Mir87] and A-TREAT [Han92] materialize only the selection nodes. The A-TREAT model chooses which node to materialize using a selectivity based heuristics. Only [FRS93] uses a global static analysis of the rules to discover the set of "most profitable" relational subexpressions to cache.

The problem of materializing relational expressions has been extensively studied in the context of view maintenance. For a recent survey on view maintenance see [GM95]. In [RSS96] the authors optimize the incremental maintenance of a materialized view by materializing additional views. The best set of additional views is chosen based on cost information and query rewriting rules. [BCL89] proposes a method for analyzing a materialized view and pattern of updates to detect invariant expressions. This work does not deal with parameterized queries. [GMR95] considers the problem of view redefinition which is quite close to the problem of maintaining parameterized queries. The paper gives some guidelines to compute the new redefined view in an incremental manner from the old one.

[7] see the explanation of the *INnode* procedure

8 Conclusion and Future Work

In this paper, we have presented algorithmic solutions for optimizing repetitive executions of SQL triggers. We proposed a general model to represent rules, programs and their interactions and we have presented algorithms that permit to generate optimized code of triggers.

Our model abstracts the complexity of database programs by focusing on iterative regions. Furthermore, our model gives a detailed view of parameter passing and local variables. The simplicity of the model enables application programmers to directly describe their transaction programs and rules in an abstract way.

Our algorithms extract invariant subqueries along a single triggering path. Since a trigger may have several triggering paths that reach it, we may end up generating several optimized versions of code. Thus, we do not compute global invariants wrt. an entire program or a set of programs. However, this method has the following advantages. First, along a triggering path, we can exploit parameter passing to perform a finer analysis of invariants. Second, a trigger can be optimized wrt. to either, a transaction program or, a set of other triggers. In the first case, the optimized trigger version takes advantage of the invariants deduced from the propagation of the transaction context. In the second case, the optimization is valid for any triggering transactions. Third, the monitoring of optimized versions of triggers is facilitated because it only requires to manage a stack that describes the progression in a current triggering path. We have also presented a simple and extensible heuristics that given a cost model and a set of query rewriting rules, selects profitable invariants. This heuristics can use any given cost model and any predefined set of query rewriting rules. Finally, our code generation algorithm works in such a way that deleting or adding a rule does not require code recomputing.

In a near future we envision to extend the detection of invariants to a more general class of parameterized queries (such as group-by queries). We plan to use a greedy algorithm for selecting the invariants. We also believe that the heuristics may be improved by using more specific query rewriting rules. Finally we are planning to conduct extensive experiments on real applications.

Aknowledgements We would like to thank Olga Kapitskaia for her helpful comments on the earlier versions of this paper and Anthony Tomasic for the fruitful discussions that help us to greatly improve this paper.

References

[BCL89] Jose A. Blakeley, Neil Coburn, and Per-Ake Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM Trans. on Database Systems*, 14(3):369–400, 1989.

[CPM96] R. Cochrane, H. Pirahesh, and N. Mattos. Integrating Triggers and Declarative Constraints in SQL. In

Proceedings on the 22nd Conference on of Very Large Data Bases. September 1996.

[For82] C. Forgy. RETE, a fast algorithm for the many patterns many objects match problem. *J. Artificial Intelligence*, 19:17–37, 1982.

[FRS93] F. Fabret, M. Régnier, and E. Simon. An Adaptative Algorithm for Incremental Evaluation of Production Rules. In *Proc. International Conference on Very Large Databases*, Dublin, Ireland, Aug. 1993.

[FT95] P. Fraternali and L. Tanca. A Structured Approach for the Definition of the Semantics of Active Databases. *ACM Transactions On Database Systems*, December 1995.

[GM93] G. Graefe and W. J. McKenna. The Volcane Optimiser generator: Extensibility and efficient search. In *IEEE international Conference on Data Engineering*, 1993.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Dataengineering Bulletin*, 18(2), 1995. Special Issue on Materialized Views and Data Warehousing.

[GMR95] A. Gupta, I.S. Mumick, and K. Ross. Adaptating Materialized Views after redefinitions. In *Proceedings of ACM SIGMOD Internationnal Conference on Management of Data*, San Jose, May 1995.

[Han92] E. Hanson. Rule Condition Testing and Action Execution in Ariel. *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, June 1992.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. *Proc. of the ACM SIGMOD International Conference on Management of Data*, June 1996.

[jM93] W. j. McKenna. *Efficient search in extensible databse optimisation: The Volcano Optimiser generator*. Ph.d. thesis, University of Colorado, 1993.

[KP81] S. Koenig and R. Paige. A Transformational Framework for the Automatic Control of Derived Data. In *Proc. International Conference on Very Large Databases*, 1981.

[LFS97] F. Llirbat, F. Fabret, and E. Simon. Eliminating Costly Redundant Computations from SQL Trigger Execution. An extended version of this paper. At http//rodin.inria/personnes/francois.llirbat/pub.html, 1997.

[Mir87] D.P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings of the National Conference on Artificial Intelligence*, Seattle, Washington, 1987.

[RSS96] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *In Proc. of the ACM SIGMOD International Conference*, june 1996.

[SLR93] T. Sellis, C. Lin, and L. Raschid. Coupling Production Systems and Database Systems: A Homogeneous Approach. *IEEE Transaction on Knowledge and Data Engineering*, 5:240–256, April 1993.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, San Francisco, California, 1996.