Chase of recursive queries*

Nieves R. Brisaboa, Antonio Fariña, Miguel R. Luaces, and José R. Paramá

Laboratorio de Bases de Datos. Dept. de Computación. Univ. Da Coruña. Campus de Elviña s/n, 15071 A Coruña. Spain. Tf. +34 981167000. Fax. +34 981167160
brisaboa@udc.es, fari@udc.es, luaces@udc.es, parama@udc.es

Abstract. In this work, we present a semantic query optimization technique to improve the efficiency of the evaluation of a subset of SQL:1999 recursive queries.
Using datalog notation, we can state our main contribution as an algorithm that builds a program P' equivalent to a given program P, when both are applied over a database d satisfying a set of functional dependencies. The input program P is a linear recursive datalog program. The new program P' has less different variables and, sometimes, less atoms in rules, thus it is cheaper to evaluate. Using CORAL, P' is empirically shown to be more efficient than the original program.

Keywords: Recursive queries, Semantic Query Optimization.

1 Introduction

Although research in recursive queries has been carried out for the last three decades, the appearance of the SQL:1999 reaffirmed the necessity of research in this area, given that SQL:1999 includes queries with linear recursion. Previous standards of SQL did not include recursion, thus research in recursive query optimization might be able to provide the suitable algorithms, to be included in the query optimizers of the database management systems to speed up the execution of recursive queries.

Although our results are focused on the development of algorithms to be included in commercial object-relational database management systems, we use datalog syntax since it is easier to manipulate. The class of datalog programs considered in this paper can be translated into SQL:1999 queries straightforwardly.

Example 1. Let us suppose that we have a database with the following relations: $stretch(Number, From, To)$, meaning that a $flight$ with code $Number$ has a stretch from the airport $From$ to the airport To; $assig(Number, Employee)$ meaning that a certain $Employee$ is assigned to the flight $Number$.

* This work is partially supported by Grants TIC2003-06593 and TIN2006-15071-C03-03 of Ministerio de Educación y Ciencia and PGIDIT05SIN10502PR of Xunta de Galicia.

Let us consider the query that computes the relation *conn(Number, From, To, First_Officer, Purser)*, meaning that flight *Number* connects the airport *From* with the airport To, possibly using several stopovers. In addition, *conn* has the information of the *First_Officer* and the *Purser* of the flight. *conn* is the transitive closure of each flight code with some additional information about the flight crew.

$P : r_0 : conn(N, F, T, O, P) : -stretch(N, F, T), assig(N, O), assig(N, P)$
$\quad r_1 : conn(N, F, T, O, P) : -stretch(N, F, Z), assig(N, O), assig(N, P), conn(N, Z, T, O, P)$

□

P is *linear*, which means it has only one recursive atom in the body of the recursive rule. Linear programs include most real life recursive queries, then much research effort has been devoted to this class of programs (see [17] for a survey of optimization techniques for this class of programs).

In addition, P is a *single recursive program* (sirup). This implies that it has only one recursive rule. Sirups is another class of programs considered by several researchers (see [6, 8, 1] for example).

The combination of both features (like in our example), is called *linear single recursive programs* (lsirup). These programs are the programs considered in this work, and they were also studied by several works (see [18, 10, 8] for example).

An interesting approach to optimize a recursive query is to see if we can transform the query, somehow, to make the recursion "smaller" and "cheaper" to evaluate. One possibility to do that is the *semantic query optimization* that uses integrity constraints associated with databases in order to improve the efficiency of the query evaluation [5]. In our case, we use functional dependencies (fds) to optimize linear recursive datalog programs.

In this paper we provide an algorithm to optimize single recursive datalog programs under fds. *The chase of datalog programs* ($Chase_F(P)$) is a modification of an algorithm introduced by Lakshmanan and Hernández [8]. It obtains from a linear single recursive program P a program P', equivalent to P when both are evaluated over databases that satisfy a set of functional dependencies F.

The chase of a datalog program P obtains an equivalent program P', where the recursive rule may has smaller number of different variables and, less number of atoms. That is, it obtains a program where the variables (in the recursive rule) are equated among them due to the effect of fds. Moreover, those equalizations of variables, sometimes reveal that an unbounded datalog program P is in fact (due to the fds) a bounded datalog program.

Example 2. Considering the program of Example 1, let us suppose that our company decides that the first officer should act as purser as well. This imposes a constraint specifying that one flight code only has one employee assigned, that is $assig : \{1\} \rightarrow \{2\}$. The set of functional dependencies F indicates that the values of the first argument determine the values in the second position in the set of facts over the predicate $assig$. For example, the atoms $assiged(ib405, peter)$ and $assig(ib405, mary)$ violate the fd $assig : \{1\} \rightarrow \{2\}$.

Using the algorithm of the *chase of datalog programs* shown in this paper, it is possible to compute the new program $Chase_F(P)$ that we call, for short, P':

$s_0 : conn(N, F, T, O, O) : -stretch(N, F, T), assig(N, O)$
$s_1 : conn(N, F, T, O, O) : -stretch(N, F, Z), assig(N, O), conn(N, Z, T, O, O)$

There are two combined beneficial effects. First, there are 6 different variables in r_0, but only 5 in s_0. Second, the number of predicates in the bodies of the rules also decreases: 3 and 4 in r_0 and r_1, respectively, but only 2 and 3 respectively in s_0 and s_1.

The reader can observe that in this case P' is very similar to the original program, the only difference is that some variables have been equalized. This equalization comes from the fact that the database fulfills the functional dependency $assig : \{1\} \rightarrow \{2\}$. Therefore, if during the evaluation of the query, an atom, let say $assig(N, O)$, is mapped to the ground fact $assig(IB405, peter)$, then another atom $assig(N, P)$ should be mapped to the same ground fact. Observe that $assig(N, O)$ and $assig(N, P)$ have the same variable in the first position, thus by $assig : \{1\} \rightarrow \{2\}$, O and P are necessarily mapped to the same ground term. □

2 Related Work

Several strategies have been proposed to tackle the process of recursive queries. Bancilhon, Maier, Sagiv and Ullman [3] introduced a technique called *magic-sets* for rewriting linear queries taking advantage of binds between nodes in rule goal trees. There is a family of papers that try to reduce the work done by a query execution by remembering previous executions of queries that can have intermediate values useful for the current execution. These techniques are called *memoing* (see [4] for a survey).

Since in practice the great majority of recursions are linear, this class of queries has attracted much work. From a logic programming perspective, several works deal with the placement of the recursive atom in the body of the rules. "Right-linear" and "left-linear" give better performance in linear recursion than magic-sets [11].

The chase as a tool to optimize queries in the framework of datalog is also used by several researchers. Lakshmanan and Hernández [8] introduced an algorithm called *the chase of datalog programs* which is based in the use of the chase [9, 2]. Recent data models have also adopted the chase to optimize queries. In the semistructured model, it has been also used as a rewriting technique for optimizing queries [7]. Popa et. al. [14] used it to optimize queries in the framework of the object/relational model.

3 Definitions

3.1 Basic Concepts

We assume the notation and definitions of [16] and then we only define the non-standard concepts. We use $EDB(P)$ to refer to the set of EDB predicate names in a datalog program P. We denote variables in datalog programs by capital

letters, while we use lower case letters to denote predicate names. For simplicity, we do not allow constants in the programs. Let a_i be a atom, $a_i[n]$ is the term in the n^{th} position of a_i.

We say that a program P *defines* a predicate p, if p is the only IDB predicate name in the program. A *single recursive rule program (sirup)* [6] is a program that consists of exactly one recursive rule and several non-recursive rules and the program defines a predicate p. A *2-sirup* is a sirup that contains only one non-recursive rule (and one recursive rule).

A rule is *linear* if there is at most one IDB atom in its body. A *linear sirup (lsirup)* [18] is a sirup such that its rules are linear. A *2-lsirup* [18] is a 2-sirup such that its rules are linear. That is, a $2-lsirup$ is a program defining a predicate p with one non-recursive rule and one recursive rule, which has only one IDB atom in its body.

Example 3. The program of Example 1 is a *2-lsirup*. □

For the sake of simplicity, many of the definitions will apply to $2-lsirups$ although the algorithm presented in this paper is valid for $lsirups$ as well. In addition, from now on, we denote with r_0 the non-recursive rule in a $2-lsirup$, and r_1 to denote the recursive rule.

Let P be a program, let r be a rule and let d be a database. Then, $P(d)$ represents the output of P when its input is d and $r(d)$ represents the output of r when its input is d. Let F be a set of functional dependencies, $SAT(F)$ represents the set of databases that satisfies F.

Let P_1 and P_2 be programs. $P_1 \subseteq_{SAT(F)} P_2$, if $P_1(d) \subseteq P_2(d)$ for all EDBs d in $SAT(F)$. $P_1 \equiv_{SAT(F)} P_2$, if $P_1 \subseteq_{SAT(F)} P_2$ and $P_2 \subseteq_{SAT(F)} P_1$.

A substitution is a finite set of pairs of the form X_i/t_i where X_i is a variable and t_i is a term, which is either a variable or a constant, and X_i and t_i are different. The result of applying a substitution, say θ, to an atom A, denoted by $\theta(A)$, is the atom A with each occurrence of X replaced by t for every pair X/t in θ. For example, consider $\theta = \{X/a, Y/b\}$ and the atom $p(X,Y)$, then $\theta(p(X,Y))$ is $p(a,b)$. A substitution θ can be applied to a set of atoms, to a rule or to a tree to get another set of atoms, rule or tree with each occurrence X replaced by t for every X/t in θ.

4 Expansion Trees

An *expansion tree* is a description for the derivation of a set of (intensional) facts by the application of one or more rules to an extensional database. First, we start with the definition of a tree generated *by only one rule*. Let r be the rule $q :- q_1, q_2, \ldots, q_k$. Then, a tree T can be built from r as follows: the node at the root of T is q, and q has k children, q_i, $1 \leq i \leq k$. We denote this tree as $tree(r)$.

Example 4. Using the program of Example 1, Figure 1 shows $tree(r_1)$.

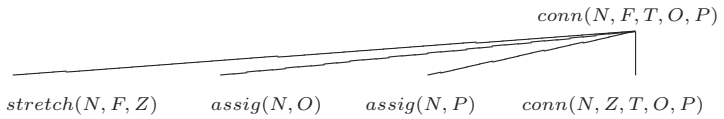


Fig. 1. $tree(r_1)$.

□

In order to be a complete expansion tree, that is, an expansion tree describing the complete execution of a program, the tree should start with the application of a non-recursive rule.

Let S and T be two trees. Then, S and T are *isomorphic*, if there are two substitutions θ and α such that $S = \theta(T)$ and $T = \alpha(S)$.

The variables appearing in the root of a tree T are called *the distinguished variables of* T. All other variables appearing in atoms of T that are different from the distinguished variables of T are called *non-distinguished variables of* T.

Let S and T be two trees, where h_t denotes the head (the node at the root) of T. Assume that exactly one of the leaves of S is an IDB atom[1], denoted by p_s. The *expansion (composition)* of S with T, denoted by $S \circ T$ is defined if there is a substitution θ, from the variables in h_t to those in p_s, such that $\theta(h_t) = p_s$. Then, $S \circ T$ is obtained as follows: build a new tree, isomorphic to T, say T', such that T' and T have the same distinguished variables, but all the non-distinguished variables of T' are different from all of those in S. Then, substitute the atom p_s in the last level of S by the tree $\theta(T')$.

From now on, we use the expression $tree(r_j \circ r_i)$ to denote $tree(r_j) \circ tree(r_i)$ and, $tree(r_j^k)$ to denote the composition of $tree(r_j)$ with itself, k times. Given a $2{-}lsirup\ P = \{r_0, r_1\}$, T_i denotes the tree $tree(r_1^i \circ r_0)$. T_i is a *complete* expansion tree since it describes the derivation of a set of IDB facts from an extensional database. Obviously, since P is a recursive program, we may construct infinitely many trees considering successive applications of the recursive rule. We call $trees(P)$ the infinite ordered collection of trees $\{T_0, T_1, T_2, T_3, \ldots\}$.

Example 5. Using the program of Example 1, Figure 2 shows T_2.

From now on, we shall consider only complete expansion trees. For the sake of simplicity we shall refer to expansion trees simply as trees.

Let T be a tree. *The level of an atom in* T is defined as follows: the root of T is at level 0, the level of an atom n of T is one plus the level of its parent. *Level* j of T is the set of atoms of T with level j. The last level of a tree T_i is the level $i + 1$. We say that two levels i and k (in a tree T_j) are *separated by* w *levels* if $|i - k| = w$ and $i \leq j + 1$ and $k \leq j + 1$.

[1] That is, the case of the trees generated by *lsirups*, since in the recursive rule of such programs, there is only one IDB predicate.

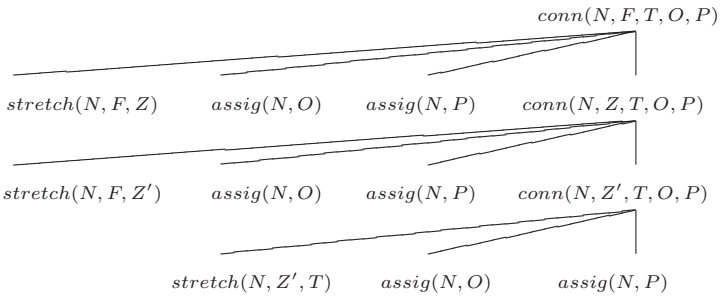


Fig. 2. The tree T_2 using the program of Example 1.

4.1 TopMost and frontier of a tree

TopMost and frontier of a tree are two rules that can be extracted from any tree. Let T be a tree: *the frontier of* T (also known as *resultant*), denoted by $frontier(T)$, is the rule $h \ :- \ l_1, \ldots, l_n$, where h is the root of T and $l_1, \ldots, l_n$ is the set of leaves of T; *the topMost of* T, denoted by $topMost(T)$, returns the rule $h \ :- \ c_1, \ldots, c_n$, where h is the root of T and $c_1, \ldots, c_n$ is the set of atoms that are the children of the root.

Example 6. Using the tree of Figure 2:

$frontier(T_2)$: $conn(N, F, T, O, P)$:- $stretch(N, F, Z), assig(N, O), assig(N, P)$
$stretch(N, F, Z'), assig(N, O), assig(N, P)$ $stretch(N, Z', T), assig(N, O), assig(N, P)$

$topMost(T_2)$:$conn(N, F, T, O, P)$:- $stretch(N, F, Z), assig(N, O), assig(N, P), conn(N, Z, T, O, P)$
□

Observe that the frontier of a tree in $trees(P)$ is a non-recursive rule, while the topMost may be a recursive one. Let P be a $2 - lsirup$, d an extensional database and T a tree in $trees(P)$. $T(d)$ represents the result of applying the rules used to build T to the input extensional database d in the order specified by T. That is, $T(d)$ can be seen or computed as $frontier(T)(d)$[2]. Let T and Q be two trees, $T \equiv_{SAT(F)} Q$ means that $T(d) = Q(d)$ for any extensional database d in $SAT(F)$.

5 Chase of a tree

The chase [9, 2] is a general technique that is defined as a nondeterministic procedure based on the successive application of dependencies (or generalized dependencies) to a set of tuples (that can be generalized to atoms).

Let us consider the following: Let F be a set of fds defined over $EDB(P)$, for some program P. Let T_i be a tree in $trees(P)$. Let $f = p : \{n\} \rightarrow \{m\}$ be a fd in F. Let q_1 and q_2 be two atoms in the leaves of T_i such that the predicate

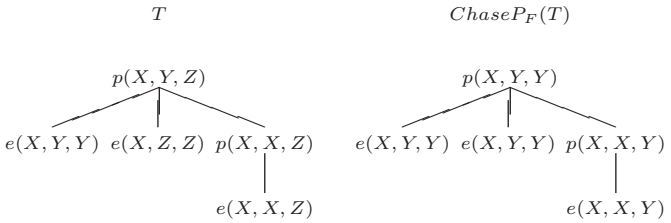
[2] Observe that if T is in $trees(P)$, the body of the frontier of a tree only contains EDB atoms.

name of q_1 and q_2 is p, $q_1[n] = q_2[n]$ and $q_1[m] \neq q_2[m]$. Note that n can be a set of positions. In addition, observe that $q_1[m]$ and $q_2[m]$ are variables since we are assuming that programs do not contain constants. An *application of the fd f to T_i* is the uniform replacement in T_i of $q_1[m]$ by $q_2[m]$ or vice versa. By uniform, we mean that $q_1[m]$ is replaced by $q_2[m]$ (or vice versa) all along the tree.

5.1 Partial Chase of a tree

The *partial chase of T with respect to F*, denoted by $ChaseP_F(T)$, is obtained by applying every fd in F to the atoms that are the leaves of T except the atoms in the last level, until no more changes can be made. Observe that although the atoms which are taken into account for the computation of the chase do not include the atoms in the last level, if a variable is renamed by the chase, such change is applied all along the tree.

Example 7. Let F be $e : \{1\} \rightarrow \{2\}$:



□

Lemma 1. *Let P be a $2 - lsirup$, let F be a set of fds over $EDB(P)$. There is a tree T_k such that for any tree T_l with $l > k$, $topMost(ChaseP_F(T_l))$ is isomorphic to $topMost(ChaseP_F(T_k))$.*

Proof Note that for all i, x such that $i > x > 0$, T_i includes all the atoms of T_{i-x} that are considered by the partial chase, then any equalization in $ChaseP_F(T_{i-x})$ is also included in $ChaseP_F(T_i)$. Therefore, there is a limit in the equalizations produced in the topMost given that all trees in $trees(P)$ with more than two levels have the same topMost, and this topMost has a finite number of variables. □

The inclusion of the last level of the tree introduces equalizations that are more difficult to model. Lemma 1 would not be true is such a case. We explored this possibility in [13].

Lemma 2. *Let P be a 2-lsirup. Let T_i be a tree in $trees(P)$, and let F be a set of fds over $EDB(P)$. Then, $T_i \equiv_{SAT(F)} ChaseP_F(T_i)$.*

The proof can be done readily, we do not include it by lack of space.

6 The Chase of datalog Programs

In [12], there is a method to find T_k, the tree such that for any tree T_l with $l > k$, $topMost(ChaseP_F(T_l))$ is isomorphic to $topMost(ChaseP_F(T_k))$. The basic idea is sketched below.

Let us consider a tree T_i in $trees(P)$ and two atoms $q_{j,k}$ and $q_{l,m}$ of T_i. $q_{j,k}$ is in the k^{th} position (numbering the atoms of its level from left to right) of level j. Similarly, $q_{l,m}$ is in the m^{th} position of level l. Now, let us suppose that the variables $q_{j,k}[n]$ and $q_{l,m}[n]$ are equalized by the $ChaseP_F(T_i)$. Then in T_{i+1}, $q_{j+1,k}[n]$ is equalized to $q_{l+1,m}[n]$ during the $ChaseP_F(T_{i+1})$. When we find a tree, say T_p, that for any equalization during its partial chase, say $q_{j,k}[n]$ equalized to $q_{l,m}[n]$, in $ChaseP_F(T_{p-r})$, where $1 \leq r \leq 2\mathcal{N}$, $q_{j-r,k}[n]$ is equalized to $q_{l-r,m}[n]$, then we have found T_k. Now the question is to find $\mathcal{N}$.

6.1 The computation of $\mathcal{N}$

To compute $\mathcal{N}$ we need to provide a previous tool.

Let P be a $2 - lsirup$. Let p_h and p_b be the IDB atoms in the head and in the body of r_1, the recursive rule of P. The *Expansion Graph of a program* P is generated with this algorithm.

1. If the arity of the IDB predicate in P is k, add k nodes named $1, \ldots, k$.
2. Add one arc from the node n to the node m, if a variable X is placed in the position n of p_h, and X is placed in the position m of p_b.
3. Add one arc from the node n without target node, if a variable X is placed in the position n of p_h, and it does not appear in p_b.
4. Add one arc without source node and target node m, if a variable X is placed in the position m of p_b and it does not appear in p_h.

Example 8. Let $P = \{r_0, r_1\}$ where r_1 contains the following IDB atoms:

$p(A, B, C, D, E, F, G, H, I, J, K, L, M) :- \ldots p(B, A, E, C, D, F, W, G, G, X, J, L, L)$

In Figure 3, we can see the expansion graph of P. □

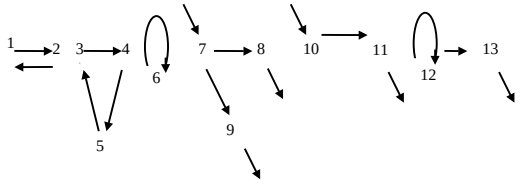


Fig. 3. Expansion Graph of P

Let G be the expansion graph of a $lsirup$ P, then $\mathcal{N}$ is the least common multiple of the number of nodes in each path in G.

Example 9. The graph in Figure 3 has $\mathcal{N}$= 6 (6= least common multiplier of 2, 3, 1, 2, 2, 2).

6.2 The algorithm

Assuming that we have found T_k, the *chase* of a $2-lsirup$ P w.r.t. a set of fds F is obtained with the algorithm shown in Figure 4.

Chase (P: a $2-lsirup$, F: a set of functional dependencies over EDB(P))
For any tree T_i with $i < k$ such that
$topMost(ChaseP_F(T_k))$ is not isomorphic to $topMost(ChaseP_F(T_i))$
Output $frontier(ChaseP_F(T_i))$;
Output $topMost(ChaseP_F(T_k))$;

Fig. 4. Chase of a datalog program.

Our algorithm is based in Lakshmanan and Hernández' algorithm [8], but our algorithm obtains better results. This improvement comes from the terminating condition. Their algorithm stops when it finds two consecutive trees with the same topMost after the partial chase. However, it is clear that after two consecutive trees with the same topMost after the partial chase, there would be bigger trees that may introduce more equalizations in the topMost after the partial chase. Our algorithm stops in a tree T_k such that it is sure that any bigger tree than T_k would not introduce any other equalization in the topMost after the partial chase. Hence, our algorithm introduces more equalities in the recursive rule of the new program.

Theorem 1. *Let P be a $2-lsirup$, let F be a set of fds over $EDB(P)$. The $Chase_F(P)$ is equivalent to P when both are evaluated over databases in $SAT(F)$.*

Proof In order to prove this theorem, we have to prove that $P' \subseteq_{SAT(F)} P$ and $P \subseteq_{SAT(F)} P'$.

We start with the proof of $P' \subseteq_{SAT(F)} P$. Let NR be the set of non-recursive rules in P', and let R be the set of recursive rules in P'. Let s be a rule in NR, by the algorithm in Figure 4, $s = frontier(ChaseP_F(T_i))$ for some tree T_i in $trees(P)$. Then, by Lemma 2, $\{s\} \subseteq_{SAT(F)} r_1^i \circ r_0$, and thus $\{s\} \subseteq_{SAT(F)} P$. Let r be a rule in R. Therefore, $r = \theta(topMost(T_j))$, where θ is the substitution defined by $ChaseP_F(T_j)$ and T_j is a tree in $trees(P)$. Since r is a recursive rule and P only has one recursive rule, then $j > 0$ and $topMost(T_j) = r_1$. Therefore, by construction, using the algorithm of the Chase of datalog programs, $r = \theta(r_1)$, and hence $r \subseteq r_1$. Thus, we have shown that for any rule r in P', $\{r\} \subseteq_{SAT(F)} P$.

Now, we tackle the other direction of the proof; $P \subseteq_{SAT(F)} P'$. We are going to prove that any fact q produced by P, when P is applied to an extensional database d in $SAT(F)$, is also produced by P', when P' is applied to d.

Let us assume that q is in $T_i(d)$, that is, q is obtained after the application to d of r_0 once, and i times r_1. We are going to prove that q is in $P'(d)$. We

prove it by induction on the number of levels of the tree T_i (in $trees(P)$) that if q is in $T_i(d)$, then q is in $P'(d)$.
Basis i=0, q is in $T_0(d)$. Then q is in $P'(d)$. Observe that P' always contains $frontier(ChaseP_F(T_0))$, since the $topMost_F(ChaseP_F(T_0))$ cannot be isomorphic to the topMost of the partial chase of any other tree in $trees(P)$ since r_0 is the only non-recursive rule of P. Then, necessarily the algorithm always outputs $frontier(ChaseP_F(T_0))$. Therefore, by Lemma 2, if q is in $T_0(d)$ then q is in $ChaseP_F(T_0)(d)$, and then q is in P'.
Induction hypothesis (IH): Let assume that $\forall q \in T_i(d)$, $1 \leq i < k$, $q \in P'(d)$.
Induction step: i=j=k. q is in $T_j(d)$. Assume q is not in any $T_m(d)$, $0 \leq m < j$, otherwise the proof follows by the IH. Thus, there is a substitution θ such that q is $\theta(p_j)$, where p_j is the root of T_j and where $\theta(t_l) \in d$ for all the leaves t_l of T_j. Therefore, q is also in $\{frontier(T_j)\}(d)$.

We have two cases: **Case 1**: $frontier(ChaseP_F(T_j))$ is one of the non-recursive rules of P'. Then by Lemma 2 q is in $P'(d)$.
Case 2: $frontier(ChaseP_F(T_j))$ is not one of the non-recursive rules of P'. Thus, by Lemma 2 $q \in \{ChaseP_F(T_j)\}(d)$ (assuming that $d \in SAT(F)$). Let γ be the substitution defined by the $ChaseP_F(T_j)$.

Let T_{sub} be the subtree of T_j that is rooted in the node at the first level of T_j that is, the recursive atom at that level. T_{sub} has one level less than T_j, therefore T_{sub} is isomorphic to T_{j-1}. Observe that this follows from the fact that in P there is only one recursive rule and one non-recursive rule.

Let q_{sub} be an atom in $T_{sub}(d)$, since T_{sub} is isomorphic to T_{j-1}, then q_{sub} is in $T_{j-1}(d)$. Hence, by IH $q_{sub} \in P'(d)$. It is easy to see that $q \in \{topMost(ChaseP_F(T_j))\}(d \bigcup q_{sub})$, that is, $q \in \{\gamma(r_1)\}(d \bigcup q_{sub})$.

By construction of P', in P' there is a rule $s_t = \theta(r_1)$, where θ is the substitution defined by the partial chase of T_k. By Lemma 1 and construction of the algorithm in Figure 4, $\gamma \equiv \theta$, otherwise $frontier(ChaseP_F(T_j))$ would be one of the non-recursive rules in P'.

We have already shown that q_{sub} is a fact in $P'(d)$. Therefore, since $s_t(d \cup q_{sub})$ obtains q, thus we have proven that if q is in $T_j(d)$ then q is also in $P'(d)$. □

7 Empirical results

We used CORAL [15], a deductive database, in order to compare the running time of the original program versus the optimized one. We ran 20 different programs over databases of different sizes. The datalog programs were synthetic queries developed by us. CORAL is an experimental system, this is a limitation, since the maximum database size is restricted, because CORAL loads all tuples in memory and then, if the database has a certain size, an overflow arises.

The computation time needed to obtain the optimized datalog program, using a program in C++ in a 200-MHz Pentium II with 48 Mbytes of RAM, takes on average 0.17 seconds with a varianza of 0.019. This is a insignificant amount of time when the database to which it is applied the query has a normal size.

The average running time of the optimized program is the 43.95% of that of the original one with a varianza of 0.10. The confidence interval of this improvement, with a confidence level of 95%, is $[28.82\%, 59.10\%]$. That is, the optimized program is between 1.7 and 3.5 times faster than the original one.

8 Conclusions and Future Work

Given a *lsirup* P and a set of fds F, we provide an algorithm that obtains a new program P' equivalent to P when both are applied over databases in $SAT(F)$. In addition, we have shown that the algorithm is correct. The algorithm shown in this paper is based in the partial chase, that does not consider atoms in the last level of the chased trees. As a future work, it would very interesting the inclusion of the last level in the computation of the chase. In that case, the chase would introduce more equalities in the alternative optimized program.

References

1. S. Abiteboul. Boundedness is undecidable for datalog programs with a single recursive rule. *Information Processing Letters*, (32):282–287, 1989.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
3. F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–16, Cambridge, Massachusetts, 24–26 Mar. 1986.
4. F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Washington, DC*, pages 16–52. ACM, May 1986.
5. U. S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kauffmann Publishers, 1988.
6. S. S. Cosmadakis and P. C. Kanellakis. Parallel evaluation of recursive rule queries. In *Proc. Fifth ACM SIGACT-SIGMOD Symposium on Principle of Database Systems*, pages 280–293, 1986.
7. A. Deutsch and V. Tannen. Reformulation of XML queries and constraints. In *ICDT*, pages 225–241, 2003.
8. L. V. S. Lakshmanan and H. J. Hernández. Structural query optimization - a uniform framework for semantic query optimization in deductive databases. In *Proc. Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principle of Database Systems*, pages 102–114, 1991.
9. D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
10. J. Naughton. Data independent recursion in deductive databases. In *Proc. Fifth ACM SIGACT-SIGMOD Symposium on Principle of Database Systems*, pages 267–279, 1986.
11. J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. *ACM SIGMOD RECORD*, 18(2), June 1989. Also published in/as: 19 ACM SIGMOD Conf. on the Management of Data, (Portland OR), May.-Jun.1989.

12. J. R. Paramá. *Chase of Datalog Programs and its Application to Solve the Functional Dependencies Implication Problem*. PhD thesis, Universidade Da Coruña, Departmento de Computación, A Coruña, España, 2001.
13. J. R. Paramá, N. R. Brisaboa, M. R. Penabad, and A. S. Places. A semantic approach to optimize linear datalog programs. *Acta Informatica*. In press.
14. L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far. In *SIGMOD*, pages 273–284, 2000.
15. R. Ramakrishnan, P. Bothner, D. Srivastava, and S. Sudarshan. Coral: A databases programming language. Technical Report TR-CS-90-14, Kansas State University, Department of Computing and Information Sciences, 1990.
16. J. D. Ullman. *Principles of Database And Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.
17. J. D. Ullman. *Principles of Database And Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.
18. M. Y. Vardi. Decidability and undecidability results for boundedness of linear recursive queries. In *Proc. Seventh ACM SIGACT-SIGMOD Symposium on Principle of Database Systems*, pages 341–351, 1988.