논 문

논문번호 20－9－14

순환 논리 질의 최적화를 위한 정적 필터의 고정점 계산
(Fixed Point Computation of Static Filters for Optimizing Recursive Logic Queries)

창 병 모* · 최 광 무** · 한 태 숙**
(Byeong-Mo Chang) (Kwang-Moo Choe) (Taisook Han)

요 약

정적 필터링은 시스템 그래프 수행을 기초로한 순환 논리 질의 최적화 방법이다. 이 방법은 수행하는 동안에 정적 필터를 이용하여 데이터의 흐름을 제한한다. 본 논문은 정적 필터의 계산 과정을 partially ordered set(poset) 위에서 하나의 변환으로 정형화하고 그 변환의 최소고정점을 구함으로써 정적 필터를 구한다. 계산된 정적 필터의 완전성(completeness)은 제시된 정형화를 이용하여 증명한다. 또한 관련 연구에 대해서 간단히 살펴보고 정적 필터링 방법과 비교한다. 본 논문에서는 정적 필터와 그 계산에 대한 정형화된 새로운 관점을 제공한다.

ABSTRACT

Static filtering is a graph-bsed optimization method for recursive logic queries based on the system graph evaluation. It restricts data-flow during evaluation by means of static filers. We formalize the computation of static filters by defining a transformation on a partially ordered set (poset) so that the least fixed point of the transformation can be the static filters. The static filters are computed by computing the least fixed point of the transformation. The completeness of the static filtering with the computed static filters is proved based on the formalism. Moreover, we review some related works in brevity and compare the static filtering with them. The formalism presented in this paper gives a new formal view to static filter and its computation.

1. Introduction

It has recently attracted much attention in the logic programming field and the deductive database field, how to answer a query effciently over a logic program. There are two basic approaches, bottom-up [2, 4, 5] and top-down[10, 14, 19, 21]. Typically, bottom-up methods deal with a set of tuples at a time, while *top-down* methods like Prolog works more naturally one tuple at a time. In case of func-

*준 회 원 : 한국과학기술원 전산학과
**종신회원 : 한국과학기술원 전산학과 교수
논문접수 : 1992년 6월 30일
심사완료 : 1993년 5월 21일

Copyright (C) 2005 NuriMedia Co., Ltd.

tion-free logic programs, bottom-up methods are guaranteed to terminate [2], while top-down methods may not. However, bottom-up methods may be inefficient if they generate many facts irrelevant to the query. A fundamental bottom-up method called *naive evaluation* tends to regenerates the same facts several times. *Seminaive evaluation* is a kind of differential approach which improves the naive evaluation by reducing the redundancy [2, 4, 5]. Based on the seminaive evaluation, a number of optimization strategies have been proposed to prevent facts irrelevant to the query from being generated [3, 4, 5, 11, 12, 13, 17].

Filtering is a graph-based optimization strategy based on the seminaive evaluation on *system graphs*, which evaluates a query in terms of the bottom-up data flow on system graphs. Filtering methods optimize the seminaive evaluation by restricting data flow through the arcs by means of filters [7, 8, 11, 12, 13]. They try to restrict data flow as much as possible. A filter is a selection attached on an arc of a system graph in order to restrict data flow through the arc. *Static filters* are computed at compile time by propagating data-independent bindings [7, 12, 13], while *dynamic filters* are computed by taking advantage of "actual tuples" generated during evaluation [8, 9, 11]. Static filtering restricts data-flow during evaluation by means of static filters [7, 12, 13], while dynamic filtering restricts data-flow by means of dynamic filters [8, 9, 11]. Static filtering is a powerful extension of the Aho-Ullman algorithm [1], (which optimizes relational queries by commuting selections with the least fixed point operator), to general recursive rules.

In this paper, we first define system graphs and the evaluation on system graphs formally We formalize the computation of static filters by defining a transformation on a partially ordered set so that the least fixed point of the transformation can be the static filters. Static filters are computed by computing the least fixed point of the transformation. We prove the completeness of the computed static filters based on the formalism. We review other related works in brevity and compare the static filtering with them. This paper gives a formal view to static filters and their computation.

This paper is organized as follows: the next section gives definitions and notations on logic programs. Section 3 describes the seminaive bottom-up evaluation on system graphs. Section 4 describes the least fixed point formalization of static filter computation. In Section 5, we prove the completeness of the static filtering. Section 6 reviews other related works in brevity and compare the static filtering with them. Section 7 summarizes results and gives further research topics.

2. Notations and Definitions

Consider well formed formulas in the first order logic. Their variables are denoted by V, X, Y and Z, terms by s and t, and atomic formula by A, B and C. An atom is said to be *ground* if no variable occurs in it A *clause* is a formula of the form

$$A_0 \leftarrow B_1, ..., B_n (n \geq 0)$$

where A_0 is an atom, and B_1, ..., B_n are atoms. A_0 is the head of the claused and B_1, ..., B_n its body. In this paper, we assume function-free and range restricted clauses If $n=0$ then the clause is simply A_0 and is called an *unit clause* or *fact*. Otherwise it is called a *rule* which are denoted by α, β, γ and δ. The number of body atoms of a rule α is denoted by n_α. A (*logic*) *program* P is a finite set of clauses. A *query* is a clause of the form $\leftarrow B_1$, ..., B_n where B_1, .., B_n are literals. In this paper we assume *function-free clauses*, which is also called Datalog.

Herbrand base B_p of a program P is the set of all atoms that are ground. Any subset of B_p is an *interpretation*. If P is a program and I is an inter-

Copyright (C) 2005 NuriMedia Co., Ltd.

pretation, T_P is a transformation associated with P which maps interpretations to interpretations.

$$T_P(I)=\{A_0 \mid A_0 \leftarrow A_1, .., A_n \text{ is a ground instance of a clause in } P, \text{ and } A_1, ..., A_n \in I\}.$$

Note that $T_P(I)$ always contains all ground instances of unit clauses in P. The fixpoint semantics determines the least fixed point of T_P, $lfp(T_P)$, as a semantics of P [20]. Since T_P is monotone, $lfp(T_P)$ exists, and it is $\cup_{m=0}^{\infty} T_P^m(\phi)$ where $T_P^{k+1}(\phi) = T_P(T_P^k(\phi))$. See [10, 20] for details.

If $A_0 \leftarrow A_1, ..., A_n$ is a ground instance of a rule α, then A_i is called a *contributor* of A_0 via α. When A_i is $p_i(\bar{\mu}_i)$ for $i=0, 1, ..., n$ where p_i is a predicate and $\bar{\mu}_i$ is a list of ground terms called a *tuple*, $\bar{\mu}_i$ is called a *contributor* of $\bar{\mu}_0$ via α. The list of the tuples $\bar{\mu}_0, \bar{\mu}_1, ..., \bar{\mu}_n$ is called a *satisfying list* of α. Let a relation $contribute_\alpha$ be defined as $\bar{\mu}_i$ $contribute_\alpha$ $\bar{\mu}_0$ if $\bar{\mu}_i$ is a contributor of $\bar{\mu}_0$ via a rule α. A tuple $\bar{\upsilon}$ is called a *far contributor* of a tuple $\bar{\mu}$ via a sequence of rules $\delta_1, ..., \delta_n$ if

$$\bar{\upsilon}_0 \; contribute_{\delta_1} \; \bar{\upsilon}_1, ..., \bar{\upsilon}_{n-1} \; contribute_{\delta_n} \; \bar{\upsilon}_n$$

for a sequence of tuples $\bar{\upsilon}_0(=\bar{\upsilon}), \bar{\upsilon}_1, ..., \bar{\upsilon}_n \; (=\bar{\mu})$ [13].

A ground atom A is *derivable* in a program P if $A \in lfp(T_P)$, that is, $A \in T_P{}^k(\phi)$ for some k. If a ground atom A is derivable and A is far contributor of some answer instance of the query, then A is said to be *useful* When a ground atom $A(=p(\bar{\mu}))$ is useful, $\bar{\mu}$ is called a useful tuple

3. System Graph and Logic Program Evaluation

We represent rules in the *AC-notation* in [17, 19] which is more suitable for data flow approach to logic program evaluation. In the *AC*-notation, all arguments of each literal are replaced by distinct variables associated with the predicate symbol of the literal. The arguments of an n-ary predicate p are replaced by the variables $P1$, ..., Pn. Since the same predicate may occur more than once in the body of a rule, a slashed variable Pj/i used to denote the j-th argument of the i-th body predicate p, so that every argument is distinct. Conditions between arguments in a rule are represented by the *conjunctive formula* (formula without disjunction) on the new variables, which is associated with that rule. A formula is defined as follows:

1. Atomic formula is *true*, *false*, or $t\theta s$ where t and s are terms and θ is $=$, $<$, or $>$.
2. If ϕ and ϕ' are formulas, $\phi \wedge \phi'$ and $\phi \vee \phi'$ are formulas.
3. A formula ϕ is said to *imply* a formula ϕ', written $\phi \rightarrow \phi'$, if every substitution satisfying ϕ satisfies ϕ'

We denote the associated conjunctive formula of a rule α by $Cond_\alpha$ in the following. A rule α is represented in the AC-notation as

$$\alpha : p(\bar{P}) \leftarrow p_1(\bar{P}_1/1), .., p_n(\bar{P}_n/n), cond_\alpha.$$

In the *AC*-notation, a query Q is represented as $Q' = ans \leftarrow Q$ with a new predicate symbol ans. A program is assumed to include its rule-form query in the following.

Example 3.1 Let us consider the following rules and query in the conventional notation.

γ : $\leftarrow p(X, a)$.
α : $p(X, Y) \leftarrow r(X. Y)$.
β : $p(X, Y) \leftarrow r(X, Z), p(Z, Y)$.

In the AC-notation, they are represented as

γ : $ans(Ans1, Ans2) \leftarrow p(P1/1, P2/1), Ans1 = P1/1, Ans2 = P2/1, P2/1 = a$
α : $p(P1, P2) \leftarrow r(R1/1, R2/1), P1 = R1/1, P2 = R2/1$.
β : $p(P1, P2) \leftarrow r(R1/1, R2/1), p(P1/2, P2/2), P1 = R1/1, R2/1 = P1/2, P2 = P2/2$.

For a rule α in the AC-notation, a function *pred*

Copyright (C) 2005 NuriMedia Co., Ltd.

(α, i) is defined to be the i-th predicate symbol in the body of α, a function *head*(α) is the predicate of the head of α. A function $Var(\alpha, i)$ is the list of all variables in the i-th literal of α, and $Var(\alpha, 0)$ is the list of all variables in the haed A *system graph* for a program is defined to be a quintuple for data flow evaluation

Definition 3.1 *A system graph for a program P is* $SG=(V_P, V_R, E_{P,R}, E_{R,P}, F)$ *where*

> V_P *and* V_R *are the sets of predicates and rules in P respectively,*
> $E_{P,R}=\{(p, \alpha)/i \mid p\in V_P,\ \alpha\in V_R,\ p=pred(\alpha, i), 1\leq i\leq n_\alpha\}$,
> $E_{R,P}=\{(\alpha, p) \mid \alpha\in V_R, p\in V_P, p=\text{head}(\alpha)\}$,
> *F is a filter function that associates a formula over* $Var(\alpha, i)$ *with each arc* $(p, \alpha)/i$ *in* $E_{P,R}$.

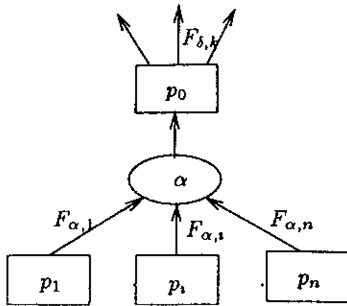


Figure 1: A part of system graph around a rule-node α

Note that there exist a number of system graphs for one program with different filter functions. We will show how to determine the filter function in the sequel The node for a predicate is called a *pred-node* and the node for a rule is a *rule-node.* The rule-node for the query is called the *query-node* in particular. Arcs are used as data ports in evaluation, so they are called *ports* Each node has its *input ports* and *output ports.* The slashed pair $(p, \alpha)/i$ in $E_{P,R}$ denotes the i-th input port of the rule-node α from the pred-node p, which is at the same time an output port of the pred-node p. An input port$(p, \alpha)/i$ is denoted by α/i as a shorthand notation since p is just determined by $pred(\alpha, i)$. A rule-node α has n_α, the number of body literals, input ports, and one output port. Figure 1 shows a system graph around a rule-node α. The set of all output ports of a pred-node p is denoted by

$$E_{P,R}[p]=\{(p, \delta)/k \mid (p, \delta)/k\in E_{P,R}\}.$$

The value of a filter function F on a port $(p, \alpha)/i$, $F((p, \alpha)/i)$, is called the filter of the port, and it represents a formula over $Var(\alpha, i)$ which tuples have to satisfy in order to pass the port. The filter $F((p, \alpha)/i)$ is denoted by $F_{\alpha,i}$ as a shorthand notation. The filter of an input port of a rule-node is called an *input filter* of the rule for simplicity. The system graph for the program in Example 1 is shown in Figure 2. The associated filters will be computed in the rest of the paper.

Algorithm simplistic simplistic seminaive evaluation shows a simplistic seminaive bottom-up evaluation on system graphs. In the algorithm, a base pred-node means a pred-node without predecessor, and a derived pred-node does a pred-node with predecessor(s). It is assumed that base pred-nodes and derived pred-nodes are disjoint, and

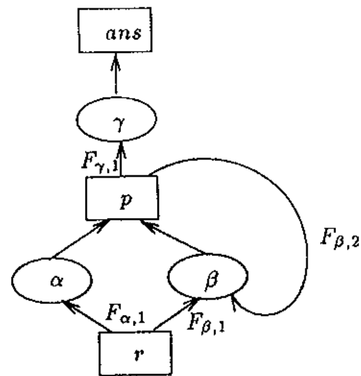


Figure 2 : System graph for the program in Example 1.

Copyright (C) 2005 NuriMedia Co., Ltd.

each base pred-node is initialized to have its facts (tuples). The evaluation is based on the hyperresolution and finds the *least fixed point* of a logic program [23, 28].

Algorithm Simplistic seminaive evaluation

1. Each base pred-node sends its tuples to its output ports.

repeat

2. Each rule-node stores new tuples received from input ports, generates new tuples from the new tuples received using its rule, and sends them to its output ports
3. Each derived pred-node stores *new* tuples from its input ports and sends them to its output ports.

until no change.

The evaluation terminates when no more new tuples can be generated by the system. It is graranteed to terminate for function-free logic programs. The completeness of this algorithm can be proved similarly to the proof of the equivalence of the least fixed point semantics and model theoretic semantics [14, 20]. The above algorithm can be considered as the evaluation on the system graph with its all filters being *true*

If the system graph has filters which are not *tru e*, a pred-node sends tuples to every output port of it if they satisfy the filter of the port. So, the filter of each port in $E_{P,R}$ restricts data flow through the port by not passing some tuples if they do not satisfy it. The evaluation on SG_P^F can be defined formally in terms of a new operator T_P^F in the following definition. An atom $B=p(t_1, ..., t_m)$is denoted by $B \in SF_{\delta,i}$ if it *satisfy* a filter $SF_{\delta,i}$.

Definition 3.2 $T_P^F : 2^{B_P} \rightarrow 2^{B_P}$

$$T_P^F(I) = \left\{ A\vartheta \;\middle|\; \begin{array}{l} C = A \leftarrow B_1, ..., B_n \in P \\ \langle B'_1, ..., B'_n \rangle \subseteq I,\ B'_i \in F_{c,i} (1 \le i \le n), \\ \vartheta = mgu(\langle B_1, ..., B_n \rangle, \langle B'_1, ..., B'_n \rangle) \end{array} \right\}$$

The seminaive evaluation on SG_P^F computes $lfp(T_P^F)$, and in case of function-free logic programs, $lfp(T_P^F) = (T_P^F)^n(\phi)$ for some finite n.

4. Computation of Static Filters

We start this section with an example which shows a computation of static filters with the idea of pushing selection in relational algebra

Example 4.1 Let us consider the program and the system graph in Example 1. Static filters can be computed with the idea of pushing selection in relational algebra.

- Pushing the selection $P2=a$ in rule-node γ results in filter $P2=a$ on port $\gamma/1$.
- Pushing the filter $P2=a$ through rule-node α results in filter $R2=a$ on $\alpha/1$.
- Pushing the filter $P2=a$ through rule-node β results in filter $P2=a$ on $\beta/2$, and filter true on $\beta/1$.

This procedure can also viewed as simulating top-down evaluation on system graph It is a basic idea of static filter computation. Static filters are computed by simulating top-down evaluation on system graph under the assumption that every body atom in a rule is executed as soon as the rule is called. When there are no more new bindings for body atoms, the simulation terminate, and the logical formula for all possible bindings of a body atom in the simulation is the static filter for that atom. Note that programs are recursive, and if the query is $\leftarrow p(a, X)$. in Example 1, pushing the selection on port $\beta/2$ generates new biding pattern of $p(Y, Z)$.

We formalize the computation of static filter as a transformation *PUSH* so that the least fixed point of *PUSH* can be the static filters The computation starts from the system graph with the filter function that associates *false* with every arc in $E_{P,R}$ (see Definition 3.1). We will define a transformation *PUSH* associated with a system graph, which

Copyright (C) 2005 NuriMedia Co., Ltd.

maps all filters in that system graph into new ones simultaneously (see Definition 4.4).

For the definition of $PUSH$, we define first a transformation $PUSH_\alpha$ associated with rule-node α, which maps (pushes) an output filter of the pred-node $head(\alpha)$ into all filters of α. $PUSH_\alpha$ is a basic operation to simulate top-down evaluation. For the denfinition of $PUSH_\alpha$, all input filters of α in a system graph $SG=(V_P, V_R, E_{P,R}, E_{R,P}, F)$ are represented as a *filter vector* $\vec{F}_\alpha$ of α which is defined as

$$\vec{F}_\alpha=<F_{\alpha,1}, ..., F_{\alpha,n_\alpha}>$$

A filter vector $\vec{F}_\alpha$ of a rule-node α is said to be less than or equal to another filter vector $\vec{F}'_\alpha$ of α, written $\vec{F}_\alpha \le \vec{F}'_\alpha$, if $F_{\alpha,i} \rightarrow F'_{\alpha,i}$ for $i=1,2,...,n_\alpha$.

We review $\bar{X}$-*consequence* in [13] for the definition of $PUSH_\alpha$. Let $\bar{X}$ be a list of some variables in a formula ϕ. A formula ϕ' is an $\bar{X}$-consequence of ϕ if $\phi \rightarrow \phi'$ and every variable in ϕ' is in $\bar{X}$. A formula ϕ' is the *most general* $\bar{X}$-consequence of ϕ if ϕ' is an $\bar{X}$-consequence of ϕ and for every other $\bar{X}$-consequence ϕ'' of ϕ, $\phi' \rightarrow \phi''$. The most general $\bar{X}$-consequence of ϕ is denoted by $\phi[\bar{X}]$. We can consider $\phi[\bar{X}]$ as the formula representing the bindings of the variables in $\bar{X}$ in ϕ (see [13]).

Denfinition 4.1 *Let α be a rule-node in a system graph $SG=(V_P, V_R, E_{P,R}, E_{R,P}, F)$. Let $F_{\delta,\kappa}$ be an output filter of the pred-node p where $p=head(\alpha)$ $PUSH_\alpha$ is a transformation associated with α which maps $F_{\delta,\kappa}$ into $\vec{F}'_\alpha= PUSH_\alpha(F_{\delta,\kappa})$ such that for each port α_i,*

$$F'_{\alpha,i}=(Cond_\alpha \wedge F_{\delta,\kappa})[Var(\alpha, i)]^1$$

$PUSH_\alpha(F_{\delta,\kappa})$ shows bindings of the body atoms when α is called by the calling patterns corresponding to $F_{\delta,\kappa}$(see Figure 1), and $F'_{\alpha,i}$ can be considered as the binding of the i-th body atom of α. If α is the query-node, we assume that $F_{\delta,\kappa}$ is *true*, so $F'_{\alpha,i}=Cond_\alpha[Var(\alpha, i)]$.

Example 4.2 Let us consider the program and the system graph in Example 1.

- If $\vec{F}_\gamma=PUSH_\gamma(true)$, then $F_{\gamma,1}=(P2=a)$.
- If $\vec{F}'_\beta=PUSH_\beta(F_{\gamma,1})$, then $F'_{\beta,1}=true$ and $F'_{\beta,2}=(R2=a)$

We extend Definition 4.1 in the following.

Definition 4.2 *Let all output filters of a pred-node p be denoted by*
$$F(E_{P,R}[p])= \bigvee_{(p,\delta)/\kappa \in E_{P,R}[p]} F_{\delta,\kappa}.$$
then, $PUSH_\alpha$ is extended to $PUSH_\alpha(F(E_{P,R}[p]))$ where $p=head(\alpha)$, in which $F'_{\alpha,i}=(Cond_\alpha \wedge F(E_{P,R}[p]))[Var(\alpha, i)]$.

It is shown in [13] that $PUSH_\alpha(F(E_{P,R}[p]))$ is equivalent to $\bigvee_{(p,\delta)/\kappa \in E_{P,R}[p]} PUSH_\alpha(F_{\delta,\kappa})$.

We define the safety of $PUSH_\alpha$ and show that $PUSH_\alpha$ is safe under the meaning.

Definition 4.3 *Let $\vec{F}'_\alpha=PUSH_\alpha(F(E_{P,R}[p])$. A filter $F'_{\alpha,i}$ in $\vec{F}'_\alpha$ is safe with respect to $F(E_{P,R}[p])$ if the condition is satisfied that if a tuple $\bar{\mu}_i$ does not satisfy $F'_{\alpha,i}$, then it can not be a contributor of some tuple satisfying $F(E_{P,R}[p])$*

In other words, if $F'_{\alpha,i}$ is *safe* with respect to $F(E_{P,R}[p])$, then $F'_{\alpha,i}$ does not prevent tuples, which contribute to some tuple satisfying $F(E_{P,R}[p])$, from being received from the port α/i.

The following lemma shows the safety of $PUSH_\alpha$, which restates Lemma 2 in [13] for the static filters. The safety of $PUSH_\alpha$ is a basis of the completeness of the static filters to be computed.

Lemma 4.1 *Let $\vec{F}'_\alpha=PUSH_\alpha(F(E_{P,R}[p]))$ where $p=head(\alpha)$ Then $F'_{\alpha,i}$ is safe with respect to $F(E_{P,R}[p])$*

1) It is assumed that $F'_{\alpha,i}$ is a formula over the variables $Var(\alpha, i)$ with slashed part deleted, so that every filter is a formula over the variables without slashed part $F_{\delta,\kappa}$ is a formula over the variables $Var(\alpha, 0)$ See Example 3

Copyright (C) 2005 NuriMedia Co., Ltd.

PROOF. The proof is by contraposition Let $\bar{\mu}_i$ *contribute* $_a \bar{\mu}_0$ for $1 \le i \le n_a$. Suppose that $\bar{\mu}_0$ satisfies $F(E_{P,R}[p])$. Then, any satisfying list $\bar{\mu}_0, \bar{\mu}_1, \ldots, \bar{\mu}_i, \ldots, \bar{\mu}_{n_a}$ satisfies $(Cond_a \wedge F(E_{P,R}[p])$ Therefore, $\bar{\mu}_i$ satisfies $F'_{a,i} = (Cond_a \wedge F(E_{P,R}[p]))[Var(a, i)]$ by the property of $\bar{X}$-consequence. □

For the computation of the static filters, we define a transformation *PUSH* associated with a system graph which maps all filters in that system graph into new ones simultaneously. All the filters in a system graph $SG = (V_P, V_R, E_{P,R}, E_{R,P}, F)$ are represented as the filter vector $\vec{F}$ of *SG* which is defined as $\vec{F} = < \vec{F}_{\delta_1}, \ldots, \vec{F}_{\delta_m} >$ where $V_R = \{\delta_1, \ldots, \delta_m\}$. The filter vector of a system graph can be considered as a representation of the filter function in that system graph, so they can be converted to each other. The set of the filter vectors corresponding to all possible filter functions for a program P is defined as

$$S(P) = \{\vec{F} \mid \vec{F} \text{ is the filter vector of a system graph for } P\}.$$

Let $\vec{F}$ and $\vec{F}'$ be two filter vectors in $S(P)$ for a program P and $V_R = \{\delta_1, \ldots, \delta_m\}$ be the set of all rule-nodes in a system graph for P Relations $\le$ on filter vectors in $S(P)$ is defined as

$$\vec{F} \le \vec{F}' \text{ if } \vec{F}'_{\delta_i} \le \vec{F}'_{\delta_i}$$

for $i = 1, 2, \ldots, m$. The relation $\le$ is a *partial order*, since it is reflexive, transitive, and antisymmetric[2]. If $\vec{F} \le \vec{F}'$ and $\vec{F}' \le \vec{F}$, then $\vec{F}$ is said to be *equivalent* to $\vec{F}'$, written $\vec{F} \equiv \vec{F}'$. A pair $(S(P), \le)$ is a poset with a unique minimum element

$$\vec{F}^{min} = << false, \ldots, false >, \ldots, < false, \ldots, false >>$$

Definition 4.4 *Let* $SG = (V_P, V_R, E_{P,R}, E_{R,P}, F)$ *be a system graph for a program P. PUSH* $: S(P) \rightarrow S(P)$ *is a transformation which maps a filter vector* $\vec{F}$ into $\vec{F}' = PUSH(\vec{F})$ *such that*

$$\vec{F}'_{\delta_1} = PUSH_{\delta_1} F(E_{P,R}[p_1])$$
$$\vec{F}'_{\delta_2} = PUSH_{\delta_2} F(E_{P,R}[p_2])$$
$$\ldots$$
$$\vec{F}'_{\delta_m} = PUSH_{\delta_1} F(E_{P,R}[p_m])$$

where $V_R = \{\delta_1, \ldots, \delta_m\}$ *and* $p_i = head(\delta_i)$ *for* $i = 1, 2, \ldots, m$

PUSH shows new bindings of the body atoms of all rules when they are called simultaneously by the calling patterns corresponding to the current filters.

Corollary 4.1 *If* $\vec{F}' = PUSH(\vec{F})$, *then every filter* $\vec{F}'_{a,i}$ *in* $\vec{F}'$ *is safe with respect to* $F(E_{P,R}[p])$ *where* $p = head(a)$.

Fixed point of *PUSH* is a filter vector $\vec{F}$ such that $\vec{F} \equiv PUSH(\vec{F})$. We compute the static filter by computing the least fixed point of *PUSH*, $lfp(PUSH)$, in the poset $(S(P), \le)$. Each filter in the least fixed point of *PUSH* represents all possible bindings of a body atom during the top-down simulation, so it is the static filter we want to find. We show with the following two lemmas that the computation of the least fixed point of *PUSH* in the poset terminates in finite time, that is, $lfp(PUSH) = PUSH^n(\vec{F}^{min})$ for some finite n.

Lemma 4.2 *The poset* $(S(P), \le)$ *for a program P satisfies the ascending chain condition that there cannot be an infinite sequence of strictly ascending elements in* $(S(P), \le)$.

PROOF. Since there are a finite number of variables and constants in a program, there are a finite number of unequivalent formulas which consist of constants and variables in the program. So there are a finite number of unequivalent filter vectors. The ascending chain condition is simply satisfied since the relation $\le$ is a partial order and there are a finite number of filter vectors. □

In the poset $(S(P), \le)$ for a program P, *PUSH* is a monotone increasing function, which is shown in the following lemma.

2) The relation $\le$ is antisymmetric in the sense that if $\vec{F} \le \vec{F}'$ and $\vec{F}' \le \vec{F}'$, then $\vec{F}' \equiv \vec{F}'$

Copyright (C) 2005 NuriMedia Co., Ltd.

Lemma 4.3 *The transformation PUSH is a monotone increasing function in the poset $(S(P), \leq)$ for a program P*

PROOF. It is shown in [13] that if a formula ϕ implies a formula ϕ', then $\phi[X]$ implies $\phi'[\overline{X}]$ where $\overline{X}$ is a set of some variables in ϕ and ϕ'. Therefore, if $\vec{F} \leq \vec{F}'$, then $PUSH(\vec{F}) \leq PUSH(\vec{F}')$ (see Definition 4.1 and Definition 4.4). □

The following theorem shows that the static filter computation terminates in finite time.

Theorem 4.1 *If $lfp(PUSH)$ is the least fixed point of PUSH in the poset $(S(P), \leq)$ for a program P with the minimal element $\vec{F}^{mn}$, then $lfp(PUSH) = PUSH^n(\vec{F}^{mn})$ for some finite n*

PROOF. By Lemma 4.2, Lemma 4.3 and Knaster-Tarski theorem [18].□

The system graph with the static filter function for a program P is defined as

$$SG^{static} = (V_P, V_R, E_{P,R}, E_{R,P}, F^{static})$$

where $\vec{F} = lfp(PUSH)$ in the poset $(S(P), \leq)$.

Example 4.3 This example illustrates the static filer computation by the least fixed point of *PUSH*. Consider the program and the system graph in Example 1. The following shows the changes of filters after each iteration of *PUSH*(see Figure 2).

Initialize: $F_{\gamma,1} = false$, $F_{\alpha,1} = false$ $F_{\beta,1} = false$. $F_{\beta,2} = false$

1st iteration: $F_{\gamma,1} = (P = a)$, $F_{\alpha,1} = false$, $F_{\beta,1} = false$, $F_{\beta\mu 2} = false$

2nd iteration: $F_{\gamma,1} = (P = a)$, $F_{\alpha\mu 1} = (P = a)$, $F_{\beta,1} = true$, $F_{\beta,2} = (P = a)$

3rd iteration: same as the 2nd iteration(the least fixed point)

The filters after the 2nd iteration are the static filters in SG^{static} for the program.

Every filter in SG^{static} satisfies the safety which is a basis of the completeness of our method.

Lemma 4.4 *Let $SG^{static} = (V_P, V_R, E_{P,R}, E_{R,P}, F^{static})$ be the system graph with the static filter function F^{static} for a program. Every filter $F^{static}_{\alpha,i}$ is safe with respect to $F^{static}(E_{P,R}[p])$ where $p = head(\alpha)$*

PROOF. By Corollary 4.1 and the definition of the fixed point. □

5. Completeness Proof

In this section, we show the completeness of the static filtering. We first define a derivation tree and proof tree for the completeness proof.

Definition 5.1 *A tree is a derivation tree of a ground atom A for a program P if.*

1) *Every vertex has a label, which is a ground instance of an atom in P*
2) *The label of the root is A*
3) *If a vertex with label B has children with labels $B_1, B_2, \cdots, B_n$, respectively, $B \leftarrow B_1, B_2, \ldots, B_n$ must be a ground instance of a program clause in P.*
4) *The label of every leaf vertex must be a fact in P*

Definition 5.2 *A proof tree for a program P is a derivation tree for a ground instance of a given query.*

When SG^{static} is the eystem graph with the static filters for a program *P*, we show the completeness of **Algorithm Simplistic seminaive evaluation** on SG^{static} in the following.

Theorem 5.1 *Let SG^{static} be the system graph with the static filter function for a program P* **Algorithm Simplistic seminaive evaluation** *on SG^{static} is complete*

PROOF. Consider any derivation tree with height k such that a ground instance $p_0(\overline{\mu}_0) \leftarrow p_1(\overline{\mu}_1), \cdots, p_n(\overline{\mu}_n)$ of α is applied at the root. We first show that if $\overline{\mu}_0$ satisfies $F(E_{P,R}[p])$ where $p = head(\alpha)$, then it is generated by the evaluation. The proof is by induction on the height of derivation trees.

Induction Basis: Derivation trees with height one

Copyright (C) 2005 NuriMedia Co., Ltd.

are simply satisfied.

Induction Hypothesis: It holds for derivation trees with height less than k

Induction Step: If $\bar{\mu}_0$ satisfies $F(E_{P,R}[p])$ where $p=head(a), \bar{\mu}_i, 1 \le i \le n_a$, satisfies $F_{a,i}$ by the safe condition of filters. By induction hypothesis, $\bar{\mu}_i$ is generated by the rule-node corresponding to the root of i-th subtree by the evaluation. Since every $\bar{\mu}_i$ is generated by the rule-node corresponding to the root of the i-th subtree by the evaluation $\bar{\mu}_0$ is generated.

For every proof tree, the tuple of the root is generated by the evaluation, since it satisfies the input filter of the query-node in SG^{static} □

6. Related Works and Comparisons

In this section, we first review dynamic filtering, Magic Set and Counting method, and compare our method with them.

To improve the performance of static filtering, we can think of taking advantage of "actual tuples" generated during evaluation. Dynamic filters are computed by propagating "actual tuples" generated during evaluation by means of *sideways informati on passing* and *backward information propagati on*. One *sideways passing graph*(SPG) per rule controls directions of sideways information passing. There are two dynamic filtering methods ever known. Dynamic filtering in [11] does not always supersede the static filtering, so that the dynamic filter is used as a conjunction with the static filter. Unfortunately it is not complete with some SPG's. We proposed an incremental dynamic filter computation based on the partial static filters in [8, 9]. The partial static filter are computed for some body atoms at compile time. They are computed with a little modification to Definition 4.1. Dynamic filters are computed incrementally from the partial static filters by means of sideways information passing and backward information propagation during evaluation. The dynamic filtering in [8, 9] is proved to be more efficient than the static filtering, and it is proved to be complete [8, 9].

Magic Sets was first described in [3] and then further developed by Beeri and Ramakrishnan [5]. The idea of Magic Sets optimization is to simulate the sideways passing of bindings of top-down evaluation like Prolog by means of the new rules introduced. The simulation by the new rules computes a set of calling patterns of each predicte during the query evaluation, which is called *magic set* Magic sets reduce the amount of potentially relevant data during the query evaluation. See [3, 4, 5] for details.

The Counting method was first described in [3] and then further developed by Sacca and Zaniolo [17] and Beeri and Ramakrishnan [5]. The idea is similar to Magic Sets, but while constructing magic sets for a query, Counting method computes the " distance" from each tuple in a magic set to the tuple of bindings specified in the query. Such magic sets are called *counting sets* Counting sets allow more precise selection while computing the query. However, Counting tends to do some extra work when the "distance" between tuples of the magic set and the query binding may not be uniquely defined.

The static filtering has a merit over other methods in the senes that it does not incur runtime overhead, while Magic Sets, Counting method and the dynamic filtering incurs some runtime overhead for computation of magic sets, counting sets and dynamic filters, respectively.

Magic Sets and Counting method can not propagate equalities (like $p(X, X)$) efficiently. They can propagate equalities only in the form of magic predicates that contain all pairs $\{< a, a >, <b, b>, ...\}$. It causes much runtime overhead. On the other hand, filtering methods (including the static filtering and dynamic filtering) propagates this equality by means of *condition* of the form $P1 = P2$, which is more efficient.

It is shown in [3, 4, 13] that none of the ap-

Copyright (C) 2005 NuriMedia Co., Ltd.

proaches (the static filtering, Magic Sets, Counting method) can not be always the best in the number of tuples generated during evaluation. It should be noted that the static filtering can be better than Magic Sets as in the following example, even though static filters are computed at compile time. Of course, Magic Sets could show more efficient behavior in some examples than the static filtering.

Example 6.1 Let us consider the following program

anc(Des, Anc, Anc)←par(Des, Anc)

anc(Des, Anc, Anc)←par(Des, Par), anc(Par, Grandpar, Anc)

When the query is ←*anc(a, Par, b)*. The static filtering computes all descendants of *b* first, and then selects tuples which satisfy the condition $A1=a$ Magic Sets, however, exhibits a less efficient behavior. It rewrite the original set of rules as follows.

magic(a, b)

magic(Par, Anc)←magic(Des, Anc), par(Des, Par)

anc(Des, Anc, Anc)←magic(Des, Anc), par(Des, Anc)

anc(Des, Anc, Anc)←magic(Des, Anc), par(Des, Par), anc(Par, Grandpar, Anc)

At first, the magic set computed by the first two rules are all pairs *magic(c, b)* where *c* is some ancestor of *a* not necessarily descending from *b*. Then it finds all the descendants of *b* who are also ancestors of *a* using the last two rules. The amount of work at this stage is the same at that done by the static filtering.

7. Concluding Remarks

We have defined the system graph, and formalized the computation of static filters as a transformation so that the least fixed point of the transformation can be the static filters. This paper gives a formal view to the static filter and its computation. We also review dynamic filtering, Magic Set and Counting method, and compare our method with them.

The static filtering can be simply extended to stratified programs (databases), which is a well defined class of logic programs with negation (see [7]). Parallel or distributed implementation of the method could be attractive further research topics. It is also a very interesting reserch topic to find heuristics or conditions to determine good SPG's, since performance of the dynamic filters could differ considerably depending on SPG's.

Acknowledgement

The first author is grateful to Prof. M. Kifer at SUNY, Stony Brook for valuable discussions on filtering. We are grateful to Prof. D-H Kim at Sungshin Women's Univ. for proofreading an earlier draft of this paper.

References

[1] A. Aho and J.D. Ullman, Universality of data retrieval languages, in: *Proc. 6th ACM Symposium on Principles of Programming Languages*, pp. 110−120, 1979.

[2] I. Balbin and K. Ramaohanarao, A differential approach to query optimization in recursive deductive databases, TR-86/7, Dept. of Computer Science, Univ. of Melbourne, 1986.

[3] F. Bancilhon, D. Maier, Y. Sagiv and J.D Ullman, Magic sets and other strange ways to implement logic programs, in: *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pp. 1−15, 1986.

[4] F. Bancilhon and R. Ramakrishnan, An amateur's introduction to recursive query processing strategies, in: *Proc. SIGMOD Int. Conf. on Management of Data*, pp. 16−

Copyright (C) 2005 NuriMedia Co., Ltd.

52, 1986.

[5] C. Beeri and R. Ramakrishnan, On the power of magic, in: *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pp. 269−283, 1987.

[6] C. Beeri, P. Kanellakis, F. Bancilhon and R. Ramakrishnan, Bounds on the propagation of selection into logic programs, in: *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pp. 214−226, 1987.

[7] B.-M. Chang, K.-M. Choe and T. Han, Static filtering for stratified programs, in revision, *Information Processing Letters.*

[8] B.-M. Chang, K.-M. Choe and T. Han, Optimized bottom-up evaluation of function-free logic programs with dynamic filtering : An incremental approach, Technical Report, CS−TR−92−66, Dept. of Computer Science, KAIST, Feb., 1992.

[9] B.-M. Chang, K.-M. Choe and T. Han, An incremental computation of dynamic filter for optimizing bottom-up evaluation of logic programs, submitted for publication, May 1992.

[10] S.W. Dietrich and D.S. Warren, Dynamic programming strategies for the evaluation of recursive queries, Technical Report TR 85−31, Computer Science Department, SUNY at Stony Brook, Sept. 1985.

[11] M. Kifer and E.L. Lozinskii, A framework for an efficient implementation of deductive database systems, in: *Proc. 6th Advanced Database Symposium, pp.* 109−116, 1986.

[12] M. Kifer and E.L. Lozinskii, Filtering data flow in deductive databases, in: *Proc. Int. Conf. on Database Theory*, pp. 186−202, 1986.

[13] M.Kifer and E.L. Lozinskii, On compile time query optimization in deductive databases by means of static filtering, *ACM Trans. Database Systems*, Vol. 15, No. 3, 1990.

[14] J.W. Lloyd, *Foundations of logic programming*, Springer-Verlag, Berlin, 1984.

[15] E.L. Lozinskii, A problem-oriented inferential database system. *ACM Trans. Database Systems*, Vol. 11, No. 3, 1986.

[16] H. Porter Ⅲ, Earley Deduction, Technical Report CS/E−86−002, Oregon Graduate Center, Mar. 1986.

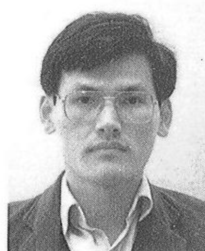
[17] D.Sacca and C. Zaniolo, The generalized counting method for recursive logic queries, *Theoretical Computer Science*, Vol. 61, pp. 1−34, 1988.

[18] A. Tarski, A lattice theoretical fixed point theorem and its applications, *Pacific Journal of Mathematics* Vol. 5, pp. 289−321, 1955.

[19] J.D. Ullman, Implementation of logical query languages for databases, *ACM Trans. Database Systems* Vol. 10, pp. 289−321, 1985.

[20] M.H. Van Emden and R.A. Kowalski, The semantics of predicate logic as a programming language, *J. ACM* Vol. 23, No. 4, 1976.

[21] L. Vieille, Recursive query processing: the power of logic, *Theoretical Computer Science* Vol. 69, pp. 1−53, 1989.



창 병 모

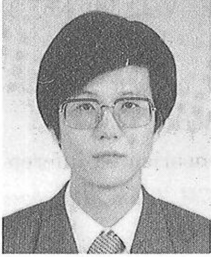
1988년 서울대학교 컴퓨터공학과 졸업(학사)

1990년 한국과학기술원 전산학과 졸업(석사)

1990년 3월부터 현재 한국과학기술원 전산학과 박사과정

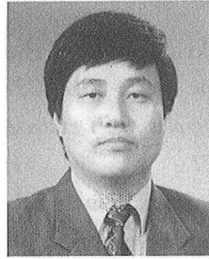
관심분야 : 논리 프로그래밍, 컴파일러 구성, 연역 데이타베이스

Copyright (C) 2005 NuriMedia Co., Ltd.



최 광 무

1976년 서울대학교 전자공학과 졸업(학사)
1978년 한국과학기술원 전산학과 졸업(석사)
1984년 한국과학기술원 전산학과 졸업(박사)
현재 한국과학기술원 전산학과 부교수
관심분야 : 프로그래밍 언어론, 논리 프로그램의 병렬 수행 및 컴파일러 구성



한 태 숙

1976년 서울대학교 전자공학과 졸업(학사)
1978년 한국과학기술원 전산학과 졸업(석사)
1990년 Univ. of North Carolina at Chapel Hill 졸업(박사)
현재 한국과학기술원 전산학과 조교수
관심분야 : 프로그래밍 언어론, 함수형 언어

Copyright (C) 2005 NuriMedia Co., Ltd.