Abstract Interpretation of Recursive Queries

Agostino Cortesi[1] and Raju Halder[2]

[1] DAIS, Università Ca' Foscari Venezia, Italy
cortesi@unive.it
[2] Dept. of Comp. Sc. & Engg., Indian Institute of Technology Patna, India

Abstract. In this paper, we extend recent works on concrete and abstract semantics of structured query languages by considering recursive queries too. We show that combining abstraction of data and widening operators that guarantee the convergence of the computation may be useful not only for static analysis purposes, but also as a sound and effective tool for query language transformations.

Keywords: Abstract Interpretation, Widening Operators, Recursive Queries, Databases.

1 Introduction

Abstract Interpretation [9] is a general theory for designing approximate semantics of programs which can be used to gather information about programs in order to provide sound answers to questions about their run-time behaviors.

Although the Abstract Interpretation is routinely used to cope with infinite state systems, it can effectively be applied to database systems which are in general finite [13,14]. In fact, due to exponentially increasing amount of data, the database systems are facing serious challenges while managing, processing, analyzing, or understanding large volume of data in restricted environments. As a result, performance of the systems in terms of optimization issues are really under big threat. The Abstract Interpretation formulation of database systems serves as a formal foundation of many interesting real-life applications, for instance, (i) to address security properties, like watermarking and access control [11,15]; (ii) to provide a novel cooperative query answering schema [12]; (iii) to serve as static analysis framework for transactions to optimize integrity constraint checking [14]; (iv) to perform abstract slicing of applications accessing or manipulating databases [16], etc. In this paper, we address a challenging feature of database query languages: the treatment of recursive queries.

Recursive queries have been introduced into the SQL languages according to ANSI SQL-99 standard in order to facilitate the access of hierarchical data, leading to more elegant and better performing applications [21]. This can be implemented by defining either a Recursive Common Table Expression (RCTE) or a Recursive View. For instance, given an initial table "`flight`" containing direct flight information, the following recursive query uses recursive common table expression "`destinations`" and returns all the destinations reachable from

C. Hota and P.K. Srimani (Eds.): ICDCIT 2013, LNCS 7753, pp. 157–170, 2013.

"Marcopolo (VCE)" along with the number of connections (including the direct ones from "`flight`") and total cost to arrive at that final destination:

```
WITH destinations (departure, arrival, connects, cost ) AS (
% anchor member.....
          SELECT f.departure,f.arrival, 0, ticket
          FROM flights f WHERE f.departure = 'Marcopolo (VCE)'
     UNION ALL
% recursive member.....
          SELECT r.departure, b.arrival, r.connects + 1, r.cost + b.ticket
          FROM destinations r, flights b WHERE r.arrival = b.departure)
SELECT DISTINCT departure, arrival, connects, cost FROM destinations;
```

Observe that the anchor member of the query is an initialization full-select that seeds the recursion, whereas the recursive member is an iterative full-select that contains a direct reference to itself in the `FROM` clause. Examples of some applications where recursive queries play important roles, are bill-of-material, reservation, trip planner, networking planning system, etc.

Over last four decades, relational algebra and relational calculus by E. F. Codd [6,7] are extensively used as a formal semantic description of query languages for relational databases. However, these models do not support the semantics of recursive queries. Later, Aho and Ulman [2] increased the expressive power of relational query languages by introducing least fix-point operator for recursive queries. Various proposals focussing on optimization of recursive queries can be found in [1,4,5,19,20].

The size and nature of database content may affect recursive query executions: for instance, a cyclic nature of data in a database may put any recursive query computation into an infinite loop. Therefore, when issuing recursive queries, it is important to the database-users to fully understand database schema and underlying database content, and to formulate recursive queries accordingly in order to avoid the possible errors which may occur during execution. For instance, if a user wants the infinite cycle to stop, she can specify the `NOCYCLE` keyword on the `CONNECT BY` clause (so that no error is issued for cyclic data) or can specify a limit on the number of iterations that a recursive query is allowed to perform.

To remedy such shortcomings and to enhance the efficiency and effectiveness of the information retrieval system, in this paper we extend our previous works on concrete and abstract semantics of database query languages [14] to the case of recursive queries too. In particular,

- We define an overapproximation of the fix-point semantics of recursive queries that replaces the possibly infinite number of concrete program executions, which might not be fully computed in practice, by an abstract execution that is guaranteed to be executable.

– We show that through a suitable choice of a widening operator [8,10], an abstract computation of recursive queries can lead to effective results, by enforcing and accelerating the convergence of fixpoint computations.

The structure of the paper is organized as follows: Section 2 illustrates a motivating example. Section 3 recalls some basics on fix-point trace semantics of programs. Section 4 describes the concrete and abstract semantics of programs embedding SQL statements that cover recursive queries too. In section 5, we define a widening operator for abstract databases that guarantees the convergence of abstract fixpoint computations. Finally, in section 6, we conclude our work.

2 A Motivating Example

Consider a database containing flight information depicted in Table 1. Recall

Table 1. Database "flight"

flight no.	source	destination	cost ($)
F001	Fiumicino (FCO)	Zurich (ZRH)	410.57
F002	Marcopolo (VCE)	Orly (ORY)	325.30
F003	Orly (ORY)	Zurich (ZRH)	200.00
F004	Orly (ORY)	Fiumicino (FCO)	428.28
F005	Zurich (ZRH)	Marcopolo (VCE)	250.15

the recursive query mentioned in the introduction part that returns all the flight destinations reachable from "Marcopolo (VCE)" along with the number of connections and total cost to arrive at that final destination. The anchor member of the query returns the base result set that contains initial destinations having direct flight from "Marcopolo (VCE)" as shown below:

departure	arrival	connects	cost
Marcopolo (VCE)	Orly (ORY)	0	325.30

In the first iteration the recursive member adds the following result set:

departure	arrival	connects	cost
Marcopolo (VCE)	Zurich (ZRH)	1	525.30
Marcopolo (VCE)	Fiumicino (FCO)	1	753.58

Continuing this way, after only three iterations, the result set returned by the running query is:

departure	arrival	connects	cost
Marcopolo (VCE)	Orly (ORY)	0	325.30
Marcopolo (VCE)	Zurich (ZRH)	1	525.30
Marcopolo (VCE)	Fiumicino (FCO)	1	753.58
Marcopolo (VCE)	Marcopolo (VCE)	2	775.45
Marcopolo (VCE)	Madrid-Barajas (MAD)	2	1164.15
Marcopolo (VCE)	Orly (ORY)	3	1100.75
Marcopolo (VCE)	Marcopolo (VCE)	3	1414.30

Observe that several pairs (departure, arrival) are repeated with different connectivity and cost. The recursive computation is, therefore, infinite due to the cyclic nature of the data in the database and can never reach to a fixpoint.

This yields the system to fail the execution and generates an "error" or "warning" message without providing any result to the end users, which might be interested only on the reachability between two airports on minimum number of connections and on the best price.

As a way to overcome such situation, in section 5 we will describe how to use an abstract version of this computation and how to get a computable sound overapproximation of all the concrete tuples generated by the concrete recursive query through a suitable widening operator.

3 Preliminaries

A fix-point of a function $f \in \mathcal{D} \to \mathcal{D}$ is an element $x \in \mathcal{D}$ such that $f(x) = x$. If f is defined over a partial order $\langle \mathcal{D}, \sqsubseteq \rangle$, then the least fix-point $\mathrm{lfp} f$ of f is the unique $x \in \mathcal{D}$ such that $f(x) = x$ and $\forall y \in \mathcal{D}.\ y = f(y) \implies x \sqsubseteq y$. Given a function f defined over a poset $\langle \mathcal{D}, \sqsubseteq \rangle$ and an element $z \in \mathcal{D}$, we denote with $\mathrm{lfp}_z^{\sqsubseteq} f$, the least fix-point of f *w.r.t.* the order $\sqsubseteq$ larger than z, if it exists. Sometimes, when the order and the element are clear from the context, we will simply write $\mathrm{lfp} f$. The dual notion is that of greatest fix-point $\mathrm{gfp} f$.

Theorem 1 (Tarski's fixpoint Theorem [3]). *Let $\langle \mathcal{D}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be a complete lattice. Let $f \in \mathcal{D} \to \mathcal{D}$ be a monotonic function. Then the set $\mathcal{L} = \{x \in \mathcal{D} \mid f(x) = x\}$ is a non-empty complete lattice w.r.t the order $\sqsubseteq$ with infimum lfp $f = \sqcap\{x \in \mathcal{D} \mid f(x) \sqsubseteq x\}$ and supremum gfp $f = \sqcup\{x \in \mathcal{D} \mid f(x) \sqsupseteq x\}$.*

3.1 Traces

Program executions are recorded in finite or infinite sequences of states, called traces, over a given set of commands. Given a (possibly infinite) set Σ of states, we consider the following sets:

$\tau^n = [0, n-1] \mapsto \Sigma$	is the set of finite sequences $\sigma = \sigma_0 \dots \sigma_{n-1}$ of length $\lvert\sigma\rvert = n \in \mathbb{N}$ over Σ.
$\tau^+ = \bigcup_{n>0} \tau^n$	is the set of non-empty finite sequences over Σ.
$\tau^* = \tau^+ \cup \{\boldsymbol{\epsilon}\}$	is the set of finite sequences over Σ where $\{\boldsymbol{\epsilon}\}$ is an empty sequence.
$\tau^\omega = \mathbb{N} \mapsto \Sigma$	is the set of infinite sequences $\sigma = \sigma_0 \dots \sigma_n \dots$ of length $\lvert\sigma\rvert = \omega$.
$\tau^\infty = \tau^+ \cup \tau^\omega$	is the set of non-empty sequences over Σ.
$\tau^\propto = \tau^\infty \cup \{\boldsymbol{\epsilon}\}$	is the set of all sequences over Σ.

The semantics of a program P can be expressed by a fixpoint operator as follows:

Theorem 2 (Fixpoint partial trace semantics [18]). *Let Σ be a set of states, $\xrightarrow{P}\subseteq \Sigma\times\Sigma$ be the transition relation associated with a program P, $\mathcal{I}_0 \subseteq \Sigma$ be a set of initial states and $\mathcal{F} \in \wp(\Sigma) \mapsto \wp(\tau^*) \mapsto \wp(\tau^*)$ be*

$$\mathcal{F}(\mathcal{I}) = \lambda X.\mathcal{I} \cup \{\sigma_0 \to \ldots \sigma_n \to \sigma_{n+1} \mid \sigma_0 \to \ldots \sigma_n \in X, \sigma_n \xrightarrow{P} \sigma_{n+1}\}.$$

Then, the partial trace semantics of P, $\mathscr{T}[\![P]\!] \in \wp(\Sigma) \mapsto \wp(\tau^*)$ is

$$\mathscr{T}[\![P]\!](\mathcal{I}_0) = lfp^{\subseteq}_{\emptyset}\mathcal{F}(\mathcal{I}_0) = \bigcup_{i\leq\omega} \mathcal{F}^i(\mathcal{I}_0)$$

4 Programs Embedding SQL Statements

In this section, we recall some basics of [14], where we introduced a concrete and abstract semantics of database query languages, and we extend it to consider also recursive queries.

An application embedding SQL statements basically interacts with two worlds: *user world* and *database world*. Corresponding to these two worlds there exist two sets of variables: database variables $\mathbb{V}_d$ and application variables $\mathbb{V}_a$. Variables from $\mathbb{V}_d$ are involved only in the SQL statements, whereas variables in $\mathbb{V}_a$ may occur in all types of instructions of the application. Any SQL statement Q is denoted by a tuple $Q = \langle A, \phi\rangle$ where A and ϕ refer to *action part* and *pre-condition part* of Q respectively. A SQL statement Q first identifies an active data set from the database using the pre-condition ϕ, and then performs the appropriate operations on that data set using the SQL action A. The pre-condition ϕ appears in SQL statements as a well-formed formula in first-order logic. Table 2 depicts the syntactic sets and the abstract syntax of programs embedding SQL statements. For more details, the reader may refer to [13,14].

4.1 Program Environments

The SQL embedded program P acts on a set of constants $const(P) \in \wp(\mathbb{C})$ and set of variables $var(P) \in \wp(\mathbb{V})$, where $\mathbb{V} \triangleq \mathbb{V}_d \cup \mathbb{V}_a$. These variables take their values from semantic domain $\mathfrak{D}_\mho$, where $\mathfrak{D}_\mho = \{\mathfrak{D} \cup \{\mho\}\}$ and $\mho$ represents the undefined value.

Now we define two environments $\mathfrak{E}_d$ and $\mathfrak{E}_a$ corresponding to the database and application variable sets $\mathbb{V}_d$ and $\mathbb{V}_a$ respectively.

Definition 1 (Application Environment). *An application environment $\rho_a \in \mathfrak{E}_a$ maps a variable $v \in dom(\rho_a) \subseteq \mathbb{V}_a$ to its value $\rho_a(v)$. So, $\mathfrak{E}_a \triangleq \mathbb{V}_a \longmapsto \mathfrak{D}_\mho$.*

Definition 2 (Database Environment). *We consider a database as a set of indexed tables $\{t_i \mid i \in I_x\}$ for a given set of indexes I_x. We define database environment by a function ρ_d whose domain is I_x, such that for $i \in I_x$, $\rho_d(i) = t_i$.*

Table 2. Abstract Syntax of programs embedding SQL statements

Syntactic Sets:

$n : \mathbb{Z}$ (Integer)	$b : \mathbb{B}$ (Boolean Expressions)
$k : \mathbb{S}$ (String)	$A : \mathbb{A}$ (Action)
$c : \mathbb{C}$ (Constants)	$\tau : \mathbb{T}$ (Terms)
$v_a : \mathbb{V}_a$ (Application Variables)	$a_f : \mathbb{A}_f$ (Atomic Formulas)
$v_d : \mathbb{V}_d$ (Database Variables)	$\phi : \mathbb{W}$ (Pre-condition)
$v : \mathbb{V} \triangleq \mathbb{V}_d \cup \mathbb{V}_a$ (Variables)	$Q : \mathbb{Q}$ (SQL statements)
$e : \mathbb{E}$ (Arithmetic Expressions)	$I : \mathbb{I}$ (Program statements)

Abstract Syntax:

$c ::= n \mid k$
$e ::= c \mid v_d \mid v_a \mid op_u\ e \mid e_1\ op_b\ e_2$, where $op_u \in \{+, -\}$ and $op_b \in \{+, -, *, /, \ldots ...\}$
$b ::= e_1\ op_r\ e_2 | \neg b | b_1 \vee b_2 | b_1 \wedge b_2 | true | false$, where $op_r \in \{=, \geq, \leq, <, >, \neq, \ldots\}$
$\tau ::= c \mid v_a \mid v_d \mid f_n(\tau_1, \tau_2, ..., \tau_n)$, where f_n is an n-ary function.
$a_f ::= R_n(\tau_1, \tau_2, ..., \tau_n) \mid \tau_1 = \tau_2$, where $R_n(\tau_1, \tau_2, ..., \tau_n) \in \{true, false\}$
$\phi ::= a_f \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x_i\ \phi \mid \exists x_i\ \phi$
$g(\boldsymbol{e}) ::= \texttt{GROUP BY}(\boldsymbol{e}) \mid id$
$r ::= \texttt{DISTINCT} \mid \texttt{ALL}$
$s ::= \texttt{AVG} \mid \texttt{SUM} \mid \texttt{MAX} \mid \texttt{MIN} \mid \texttt{COUNT}$
$h(e) ::= s \circ r(e) \mid \texttt{DISTINCT}(e) \mid id$
$h(*) ::= \texttt{COUNT}(*)$
$\boldsymbol{h}(\boldsymbol{x}) ::= \langle h_1(x_1), ..., h_n(x_n) \rangle$, where $\boldsymbol{h} = \langle h_1, ..., h_n \rangle$ and $\boldsymbol{x} = \langle x_1, ..., x_n \rangle$
$f(\boldsymbol{e}) ::= \texttt{ORDER BY ASC}(\boldsymbol{e}) \mid \texttt{ORDER BY DESC}(\boldsymbol{e}) \mid id$
$A ::= select(v_a,\ f(\boldsymbol{e'}),\ r(\boldsymbol{h}(\boldsymbol{x})),\ \phi,\ g(\boldsymbol{e})) \mid update(\boldsymbol{v_d},\ \boldsymbol{e}) \mid insert(\boldsymbol{v_d},\ \boldsymbol{e}) \mid delete(\boldsymbol{v_d})$
$Q ::= \langle A, \phi \rangle \mid Q'\ \texttt{UNION}\ Q'' \mid Q'\ \texttt{INTERSECT}\ Q'' \mid Q'\ \texttt{MINUS}\ Q''$
$I ::= skip \mid v_a := e \mid v_a :=? \mid Q \mid if\ b\ then\ I_1\ else\ I_2 \mid while\ b\ do\ I \mid I_1; I_2$

Definition 3 (Table Environment). *Given a database environment ρ_d and a table $t \in d$. We define $attr(t) = \{a_1, a_2, ..., a_k\}$. So, $t \subseteq D_1 \times D_2 \times .... \times D_k$ where, a_i is the attribute corresponding to the typed domain D_i . A table environment ρ_t for a table t is defined as a function such that for any attribute $a_i \in attr(t)$,*

$$\rho_t(a_i) = \langle \pi_i(l_j) \mid l_j \in t \rangle$$

Where π is the projection operator, i.e. $\pi_i(l_j)$ is the i^{th} element of the l_j-th row. In other words, ρ_t maps a_i to the ordered set of values over the rows of the table t.

4.2 Small-Steps Operational Semantics of Programs Embedding SQL

Let Σ be a set of states defined by $\Sigma \triangleq \mathfrak{E}_d \times \mathfrak{E}_a$, where $\mathfrak{E}_d$ and $\mathfrak{E}_a$ denote the set of all database environments and the set of all application environments respectively. That is, a state $\sigma \in \Sigma$ is denoted by a tuple $\langle \rho_d, \rho_a \rangle$ where $\rho_d \in \mathfrak{E}_d$ and $\rho_a \in \mathfrak{E}_a$ are the database environment and application environment respectively. The set of states of P is, thus, defined as $\Sigma[\![P]\!] \triangleq \mathfrak{E}_d[\![P]\!] \times \mathfrak{E}_a[\![P]\!]$ where $\mathfrak{E}_d[\![P]\!]$ and $\mathfrak{E}_a[\![P]\!]$ are the set of database and application environments of the program P whose domain is the set of program variables.

The transition relation $S \in (\mathbb{I} \times \Sigma) \mapsto \wp(\Sigma)$ specifies which successor states $\langle \rho'_d, \rho'_a \rangle \in \Sigma$ can follow when a statement $I \in \mathbb{I}$ executes on state $\langle \rho_d, \rho_a \rangle \in \Sigma$.

Therefore, the transitional semantics $S[\![P]\!] \in (P \times \Sigma[\![P]\!]) \mapsto \wp(\Sigma[\![P]\!])$ of a program $P \in \mathbb{P}$ restricts the transition relation to program instructions, *i.e.*

$$S[\![P]\!](\langle \rho_d, \rho_a \rangle) = \{\langle \rho'_d, \rho'_a \rangle \mid \langle \rho_d, \rho_a \rangle, \langle \rho'_d, \rho'_a \rangle \in \Sigma[\![P]\!] \wedge \langle \rho'_d, \rho'_a \rangle \in S[\![I]\!](\rho_d, \rho_a) \wedge I \in P\}$$

4.3 Partial Trace Semantics of Programs Embedding SQL

Let us consider a set of states in the form of $\Sigma_{\mathcal{L}} = \mathcal{L} \times \Sigma$ where $\mathcal{L}$ is the set of labels. Let $\mathcal{I}_0 = \{(l_0, \rho_0) \mid \rho_0 = \langle \rho_d, \rho_a \rangle \in \Sigma[\![P]\!]\} \subseteq \Sigma_{\mathcal{L}}[\![P]\!]$ be the set of initial states of P where l_0 is the entry label of P. According to Theorem 2, the partial trace semantics of P can be defined as

$$\mathscr{T}[\![P]\!](\mathcal{I}_0) = \mathrm{lfp}^{\subseteq}_{\emptyset} \mathcal{F}(\mathcal{I}_0) = \bigcup_{i \leq \omega} \mathcal{F}^i(\mathcal{I}_0)$$

where

$$\mathcal{F}(\mathcal{I}) = \lambda \mathcal{T}.\, \mathcal{I} \cup \{(l_0, \rho_0) \to \ldots (l_n, \rho_n) \to (l_{n+1}, \rho_{n+1}) \mid (l_0, \rho_0) \to \ldots (l_n, \rho_n) \in \mathcal{T} \\ \wedge l_n = \mathcal{L}[\![I]\!] \wedge \rho_{n+1} \in S[\![I]\!](\rho_n) \wedge l_{n+1} \in \mathtt{succ}(l_n)\}$$

and the function $\mathtt{succ}(l)$ returns the set of program labels which are successors of l.

Partial Trace Semantics of Recursive Queries. Let us now turn our attention to recursive queries. Let $Q = \langle {}^{l_0}Q_a, {}^{l_1}Q_r \rangle$ be a recursive query where ${}^{l_0}Q_a$ and ${}^{l_1}Q_r$ denote the anchor member and the recursive member in Q at program labels l_0 and l_1 respectively. Suppose $\mathcal{I}_0 = \{\langle l_0, \rho_0 \rangle \mid \rho_0 = \langle \rho_d, \rho_a \rangle \in \Sigma[\![Q]\!]\}$ is the set of initial states of Q such that the recursive common table expression is $\emptyset$. Let $S[\![Q_a]\!](\langle l_0, \rho_0 \rangle) = \langle l_1, \rho_1 \rangle$. By specializing the definition above, the partial trace semantics of Q can be defined by

$$\mathscr{T}[\![Q]\!](\mathcal{I}_0) = \mathrm{lfp}^{\subseteq}_{\emptyset} \mathcal{F}(\mathcal{I}_0) = \bigcup_{i \leq \omega} \mathcal{F}^i(\mathcal{I}_0)$$

where

$$\mathcal{F}(\mathcal{I}) = \lambda \mathcal{T}.\, \mathcal{I} \cup \{(l_0, \rho_0) \to (l_1, \rho_1) \to \ldots (l_1, \rho_n) \to (l_1, \rho_{n+1}) \mid \\ ((l_0, \rho_0) \to (l_1, \rho_1) \to \ldots (l_1, \rho_n)) \in \mathcal{T} \wedge \rho_{n+1} \in S[\![Q_r]\!](\rho_n)\}$$

Approximation of Fix-Point Semantics. As stated in [10], when considering an abstraction of the semantic domain, an overapproximation of the fix-point semantics depicted so far can be efficiently computed.

Let $\mathcal{D}$ be the semantic domain. Let $\mathrm{lfp}_{\perp} \mathcal{F}$ where $\mathcal{F} \in \mathcal{D} \mapsto \mathcal{D}$ be the semantics of a given program P. Assume the Galois Connection $\langle (\mathcal{D}, \mathcal{F}, \perp), \alpha, \gamma, (\widetilde{\mathcal{D}}, \widetilde{\mathcal{F}}, \widetilde{\perp}) \rangle$ where α and γ are abstraction and concretization functions respectively, such that $\mathrm{lfp}^{\widetilde{\perp}}(\widetilde{\mathcal{F}})$ is not computable iteratively in finitely many steps. Suppose $\widetilde{\mathcal{A}}$ is an upper approximation of $\mathrm{lfp}^{\widetilde{\perp}}(\widetilde{\mathcal{F}})$ which is effectively computed using widening [8,10]. Since $\mathrm{lfp}_{\perp} \mathcal{F} \sqsubseteq \mathrm{lfp}^{\widetilde{\perp}}(\widetilde{\mathcal{F}})$ and $\mathrm{lfp}^{\widetilde{\perp}}(\widetilde{\mathcal{F}}) \sqsubseteq \widetilde{\mathcal{A}}$, so by monotonicity and transitivity, we get $\mathrm{lfp}_{\perp} \mathcal{F} \sqsubseteq \widetilde{\mathcal{A}}$.

4.4 Lifting the Semantics to Abstract Domains

We lift the concrete semantics of programs embedding SQL statements to an abstract setting by introducing the notion of abstract databases in which some information are disregarded and concrete values are possibly represented by suitable abstractions.

Definition 4 (Abstract Databases). *Let dB be a database. A database $\widetilde{dB} = \alpha(dB)$ where α is the abstraction function, is said to be an abstract version of dB if there exist a representation function γ, called concretization function such that for all tuple $\langle x_1, x_2, \ldots, x_n\rangle \in dB$ there exist a tuple $\langle \widetilde{y}_1, \widetilde{y}_2, \ldots, \widetilde{y}_n\rangle \in \widetilde{dB}$ such that $\forall i \in [1 \ldots n]$ $(x_i \in id(\widetilde{y}_i) \vee x_i \in \gamma(\widetilde{y}_i))$, where id represents identity function.*

Example 1. Consider the table "*emp*" depicted in Table 3 that provides information about employees of a company. We assume that ages, salaries, and number of children of employees lie between 5 to 100, between 500 to 10000 and between 0 to 10 respectively. Considering an abstraction where ages and salaries of employees are abstracted by elements from the domain of intervals, and number of children in the attribute '*Child-no*' are abstracted by abstract values from the abstract domain $D^{abs}_{Child-no}$={$\bot$, Zero, Few, Medium, Many, $\top$} where $\top$ represents "*any*" and $\bot$ represents "$undefined$". The abstract table "$\widetilde{emp}$" corresponding to "*emp*" *w.r.t.* these abstractions is shown in Table 4. The

Table 3. Employee table "*emp*"

eID	$Name$	Age	Dno	Pno	Sal	$Child-no$
1	Matteo	30	2	1	2000	4
2	Alice	22	1	2	1500	2
3	Joy	50	2	3	2300	3
4	luca	10	1	2	1700	1
5	Deba	40	3	4	3000	5
6	Andrea	70	1	2	1900	2
7	Alberto	18	3	4	800	1
8	Bob	14	2	3	4000	3

Table 4. Abstract table "$\widetilde{emp}$" corresponding to "*emp*"

$\widetilde{eID}$	$\widetilde{Name}$	$\widetilde{Age}$	$\widetilde{Dno}$	$\widetilde{Pno}$	$\widetilde{Sal}$	$\widetilde{Child-no}$
1	Matteo	[25,59]	2	1	[1500,2499]	Medium
2	Alice	[12,24]	1	2	[1500,2499]	Few
3	Joy	[25,59]	2	3	[1500,2499]	Medium
4	luca	[5,11]	1	2	[1500,2499]	Few
5	Deba	[25,59]	3	4	[2500,10000]	Many
6	Andrea	[60,100]	1	2	[1500,2499]	Few
7	Alberto	[12,24]	3	4	[500,1499]	Few
8	Bob	[12,24]	2	3	[2500,10000]	Medium

correspondence between concrete and abstract values of the attribute, for instance '*Child-no*' can be formally expressed by the abstraction and concretization functions $\alpha_{child-no}$ and $\gamma_{child-no}$ respectively as follows:

$$\alpha_{child-no}(X) \triangleq \begin{cases} \bot & \text{if } X = \emptyset \\ \text{Zero} & \text{if } X = \{0\} \\ \text{Few} & \text{if } \forall x \in X:\ 1 \leq x \leq 2 \\ \text{Medium} & \text{if } \forall x \in X:\ 3 \leq x \leq 4 \\ \text{many} & \text{if } \forall x \in X:\ 5 \leq x \leq 10 \\ \top & \text{otherwise} \end{cases}$$

$$\gamma_{child-no}(y) \triangleq \begin{cases} \emptyset & \text{if y=}\bot \\ \{0\} & \text{if y=Zero} \\ \{x:\ 1 \leq x \leq 2\} & \text{if y=Few} \\ \{x:\ 3 \leq x \leq 4\} & \text{if y=Medium} \\ \{x:\ 5 \leq x \leq 10\} & \text{if y=Many} \\ \{x:\ 0 \leq x \leq 10\} & \text{if y=}\top \end{cases}$$

We can similarly define the abstraction-concretization functions for other attributes as well. Observe that the abstraction-concretization functions for the attributes 'eID', 'Dno' and 'Pno' are identity function id.

The abstract versions corresponding to all concrete functions such as `Group By`, `Order By`, Aggregate Functions, Set Operations, etc are defined in such a way so as to preserve the soundness condition. For more details, see [13,14].

5 Widening Operators for Abstract Databases

In section 4.3, we saw that widening operator is crucial to make the abstract semantics computable. In this section, we instantiate a widening operator to the case of abstract databases to suitably deal with convergence of abstract recursive query computations.

In the literature [8,10], a widening operator is defined as follows:

Definition 5 (Widening). *Let $\langle \mathcal{D}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be a lattice. The partial operator $\nabla:\ \mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$ is a widening if*

- *for each $x, y \in \mathcal{D}$: $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$.*
- *for each increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$, the increasing chain defined by $y_0 = x_0$, $y_{n+1} = y_n \nabla x_{n+1}$ for $n \in \mathbb{N}$, is not strictly increasing.*

Theorem 3 (Convergence [9]). *Given a sequence $\{x_0, x_1, x_2, \ldots\ldots\}$, the increasing chain $\langle y_k, k \in \mathbb{N} \rangle$ defined by $y_0 = x_0$ and*

$$y_{n+1} := \begin{cases} y_n & \text{if } \exists l \leq n:\ x_{n+1} \sqsubseteq y_l \\ \\ y_n \ \nabla\ x_{n+1}, & \text{otherwise} \end{cases}$$

is strictly increasing up to a least $l \in \mathbb{N}$ such that $x_l \sqsubseteq y_l$ and the sequence is stationary at l onwards.

The following example defines a widening operator for the domain of intervals.

Example 2 (Widening for intervals). Consider a lattice of intervals $\mathcal{D} = \{\bot\} \cup \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\},\ u \in \mathbb{Z} \cup \{+\infty\}\}$ ordered by $\forall x \in \mathcal{D},\ \bot \leq x$ and $[l_0, u_0] \leq [l_1, u_1]$ if $l_1 \leq l_0$ and $u_0 \leq u_1$. Let k be a fixed positive integer constant, and I be any set of indices. The following defines a threshold widening operator defined on $\mathcal{D}$ by

$$\nabla^k(\{\bot\}) = \bot$$

$$\nabla^k(\{\bot\} \cup S) = \nabla^k(S)$$

$$\nabla^k(\{[l_i, u_i] :\ i \in I\}) = [h_1, h_2]$$

where

$$h_1 = min\{l_i :\ i \in I\} \ \text{ if } \ min\{l_i :\ i \in I\} > -k \text{ else } -\infty$$

$$h_2 = max\{u_i :\ i \in I\} \ \text{ if } \ max\{u_i :\ i \in I\} < k \text{ else } +\infty$$

As a partial order between abstract databases $\widetilde{d_1}, \widetilde{d_2} \in \widetilde{D}$ (denoted $\widetilde{d_1} \sqsubseteq \widetilde{d_2}$) we consider the following:

Definition 6 (Partial order between abstract databases). *Given two abstract databases $\widetilde{d_1}, \widetilde{d_2} \in \widetilde{\mathcal{D}}$. The partial order between $\widetilde{d_1}$ and $\widetilde{d_2}$ (denoted $\widetilde{d_1} \sqsubseteq \widetilde{d_2}$) can be defined as follows:*

- $\widetilde{d_1} \sqsubseteq \widetilde{d_2}$ *if* $\forall \widetilde{t_i} \in \widetilde{d_1}, \exists \widetilde{t_j} \in \widetilde{d_2}\ :\ \widetilde{t_i} \leq \widetilde{t_j}$.
- $\widetilde{t_i} \leq \widetilde{t_j}$ *if* $attr(\widetilde{t_i}) \subseteq attr(\widetilde{t_j})$ *and* $\forall \widetilde{r} \in \widetilde{t_i}, \exists \widetilde{r'} \in \widetilde{t_j}\ :\ \gamma(\widetilde{r}.\widetilde{a}) \subseteq \gamma(\widetilde{r'}.\widetilde{a})$ *for all* $\widetilde{a} \in attr(\widetilde{t_i})$.

Let us consider recursive queries now. The convergence of recursive queries on an initial database depends on:

- the volume of the data in the initial database,
- the absence of cyclic data.

Note that we assume the initial database always finite as databases are finite (though unbounded in their sizes).

Let $Q :\ \mathcal{D} \mapsto \mathcal{D}$ be a recursive query which is monotone over the database domain $\mathcal{D}$. Consider a Galois Connection $(\langle \mathcal{D}, Q, d_0 \rangle, \alpha, \gamma, \langle \widetilde{\mathcal{D}}, \widetilde{Q}, \widetilde{d_0} \rangle)$, where d_0 is the initial database in which recursive common table expression is $\emptyset$ and

$\alpha :\ \mathcal{D} \to \widetilde{\mathcal{D}}$ - Abstraction mapping function.
$\gamma :\ \widetilde{\mathcal{D}} \to \mathcal{D}$ - Concretization mapping function.
$\langle \mathcal{D}, Q, d_0 \rangle$ - Concrete specifications.
$\langle \widetilde{\mathcal{D}}, \widetilde{Q}, \widetilde{d_0} \rangle$ - Abstract specification.

In case of abstract least fix-point computation $\mathrm{lfp}^{\sqsubseteq}_{\widetilde{d_0}}$ where $\sqsubseteq$ is a partial order over $\widetilde{\mathcal{D}}$, of recursive queries, either one of the following conditions is sufficient to guarantee the convergence:

- The abstraction function α removes the cyclic nature of the data when abstracting d_0 into $\widetilde{d_0}$.
- The abstract database domain $\widetilde{\mathcal{D}}$ is of finite height.

If none of the conditions is satisfied, then in order to enforce and accelerate the convergence of abstract computations of recursive queries, we are forced to instantiate a suitable widening operator.

Widening Construction. Given an abstract database domain $\widetilde{\mathcal{D}}$ over a set of attributes $\{\widetilde{a_i} \in \widetilde{A_i} \ : \ i \in \mathbb{I}\}$, where $\mathbb{I}$ is a finite set and $\widetilde{A_i}$ is the abstract attribute domain for $\widetilde{a_i}$. Let $\mathbb{J} \subseteq \mathbb{I}$ be a set of attributes such that $|\widetilde{A_j}|$ is finite for all $j \in \mathbb{J}$.

Let $\widetilde{d_1} \sqsubseteq \widetilde{d_2} \sqsubseteq \cdots \sqsubseteq \widetilde{d_n}$ be an increasing chain in $\widetilde{\mathcal{D}}$. Let ∇_i be the widening operator defined over $\widetilde{A_i}$ with $i \in \mathbb{I} \backslash \mathbb{J}$.

A generic widening operator ∇_d over the whole database domain is defined as follows: Let $\widetilde{d_1} = \{\widetilde{t_1^1}, \widetilde{t_2^1}, \ldots, \widetilde{t_n^1}\}$ and $\widetilde{d_2} = \{\widetilde{t_2^1}, \widetilde{t_2^2}, \ldots, \widetilde{t_m^2}\}$.

$$\widetilde{d_1} \ \nabla_d \ \widetilde{d_2} = ((((\widetilde{d_1} \ \nabla_t \ \widetilde{t_1^2}) \ \nabla_t \ \widetilde{t_2^2}) \ldots) \ \nabla_t \ \widetilde{t_m^2}) \tag{1}$$

where

$$\widetilde{d} \ \nabla_t \ \widetilde{t} = \begin{cases} \text{if } \ \exists \widetilde{t'} \in \widetilde{d} \ : \ attr(\widetilde{t'}) = attr(\widetilde{t}) \\ (\widetilde{d} \backslash \widetilde{t'}) \cup (\widetilde{t'} \oplus \widetilde{t}) \\ \\ \widetilde{d} \cup \{\widetilde{t}\} \quad \text{otherwise} \end{cases}$$

Consider two abstract tables $\widetilde{t}$ and $\widetilde{t'}$ where $attr(\widetilde{t}) = attr(\widetilde{t'})$. Let $\widetilde{t} = \{\widetilde{r_1}, \widetilde{r_2}, \ldots, \widetilde{r_p}\}$ and $\widetilde{t'} = \{\widetilde{r'_1}, \widetilde{r'_2}, \ldots, \widetilde{r'_q}\}$. The operation $\widetilde{t^*} = \widetilde{t} \oplus \widetilde{t'}$ is defined as follows: $\forall \widetilde{r'_l} \in \widetilde{t'}$,

$$\begin{cases} \text{if } \ \exists \widetilde{r_k} \in \widetilde{t} \ : \ \forall \widetilde{a} \in (\{\widetilde{a_j} \mid j \in \mathbb{J}\} \cap \{attr(\widetilde{t})\}) \ \ \widetilde{r_k}.\widetilde{a} = \widetilde{r'_l}.\widetilde{a} \\ \widetilde{t^*} = (\widetilde{t} \backslash \widetilde{r_k}) \cup \{\widetilde{r_k} \diamond \widetilde{r_l}\} \\ \\ \widetilde{t^*} = \widetilde{t} \cup \{\widetilde{r'_l}\} \quad \text{otherwise} \end{cases}$$

Consider two records of the same table, *i.e.* two abstract tuples $\widetilde{r}$ and $\widetilde{r'}$ such that for all $h \in \mathbb{J} \ : \ \widetilde{r}.\widetilde{a_h} = \widetilde{r'}.\widetilde{a_h}$ (where $\widetilde{a_h}$ is an attribute of that table). The operation $\widetilde{r^*} = \widetilde{r} \diamond \widetilde{r'}$ is defined as follows:

$$\widetilde{r^*}.\widetilde{a_k} = \begin{cases} \widetilde{r}.\widetilde{a_k} \ \nabla_k \ \widetilde{r'}.\widetilde{a_k} & \text{if } k \notin \mathbb{J} \\ \\ \widetilde{r}.\widetilde{a_k} & \text{otherwise} \end{cases} \tag{2}$$

where ∇_k is the widening operator on the domain $\widetilde{A_k}$ satisfying the standard conditions of definition 5.

Let us illustrate the widening operator defined above using our running example.

Example 3. Consider the running example of section 2. Consider the domain of intervals as an abstract domain for the attribute "cost". The abstract database "$\widetilde{flight}$" corresponding to "flight" is depicted in Table 5. In this example, the

Table 5. Abstract Database "$\widetilde{flight}$"

$\widetilde{flight\ no.}$	$\widetilde{source}$	$\widetilde{dest}$	$\widetilde{cost}$
F001	Fiumicino (FCO)	Zurich (ZRH)	[410.57, 410.57]
F002	Marcopolo (VCE)	Orly (ORY)	[325.30, 325.30]
F003	Orly (ORY)	Zurich (ZRH)	[200.00, 200.00]
F004	Orly (ORY)	Fiumicino (FCO)	[428.28, 428.28]
F005	Zurich (ZRH)	Marcopolo (VCE)	[250.15, 250.15]

indexed set $\mathbb{J}$ corresponds to the attributes $\widetilde{flight\ no.}$, $\widetilde{source}$ and $\widetilde{dest}$: observe that for these attributes the abstraction function is just the identity. For the other attributes $\widetilde{connects}$ and $\widetilde{cost}$, corresponding to the indexed set $\mathbb{I}\backslash\mathbb{J}$, we consider the widening operator for the domain of intervals introduced in Example 2. The application of this widening operator results into a fast convergence of the fixpoint computation of the running query, yielding to the following approximate result after $(k_1 * k_2)$ iterations, where k_1 and k_2 represent the threshold for the widening operators in case of the two interval domains corresponding to the attributes $\widetilde{connects}$ and $\widetilde{cost}$ respectively:

$\widetilde{departure}$	$\widetilde{arrival}$	$\widetilde{connects}$	$\widetilde{cost}$
Marcopolo (VCE)	Orly (ORY)	$[0, \infty]$	$[325.30, \infty]$
Marcopolo (VCE)	Zurich (ZRH)	$[1, \infty]$	$[525.30, \infty]$
Marcopolo (VCE)	Fiumicino (FCO)	$[1, \infty]$	$[753.58, \infty]$
Marcopolo (VCE)	Marcopolo (VCE)	$[2, \infty]$	$[775.45, \infty]$

Observe that although users are not able to get the exact details, they still get a precise information on the minimum number of connections and on the best price for all reachable destinations. A more sophisticated widening operator might also be used to get more precise information, by allowing the multiple occurrences of the same pair ($\widetilde{departure}$, $\widetilde{arrival}$) up to a threshold number of cases.

Convergence. It can be observed from equations 1 and 2 that ∇_d is defined in terms of widening operators that satisfy the standard conditions of Definition 5. Therefore, by Theorem 3, we can prove that the upward iteration sequence starting at the initial abstract database $\widetilde{d}_0$ and such that

$$\widetilde{d_{i+1}} := \begin{cases} \widetilde{d}_i & \text{if } \widetilde{Q}(\widetilde{d}_i) \sqsubseteq \widetilde{d}_i \\ \widetilde{d}_i \ \nabla_d \ \widetilde{Q}(\widetilde{d}_i), & \text{otherwise} \end{cases}$$

where $\widetilde{Q} : \ \widetilde{\mathcal{D}} \mapsto \widetilde{\mathcal{D}}$ is an abstract recursive query which is monotonic function, converges after a finite number $l \in \mathbb{N}$ of iterations and gives a sound approximation of $\mathrm{lfp}^{\sqsubseteq}_{\widetilde{d}_0} \widetilde{Q}$.

6 Conclusions

In this work, we dealt with a fast convergence of recursive queries independent of the database content. We discussed how the application of widening operators in the context of Abstract Interpretation make the system more effective and efficient in case of recursive query executions, in particular when the presence of large volume of data or the presence of cyclic nature of data in a database affect recursive execution and result into a convergence after finitely many steps or result into an infinite loop.

Acknowledgement. Work partially supported by PRIN 2010-11 project "Security Horizons".

References

1. Agrawal, R., Devanbu, P.: Moving selections into linear least fixpoint queries. IEEE Transactions on Knowledge and Data Engineering 1(4), 424–432 (1989)
2. Aho, A.V., Ullman, J.D.: Universality of data retrieval languages. In: Proceedings of the POPL 1979, pp. 110–119. ACM Press (1979)
3. Alfred, T.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. 5(2), 285–309 (1955)
4. Burzańska, M., Stencel, K., Wiśniewski, P.: Pushing Predicates into Recursive SQL Common Table Expressions. In: Grundspenkis, J., Morzy, T., Vossen, G. (eds.) ADBIS 2009. LNCS, vol. 5739, pp. 194–205. Springer, Heidelberg (2009)
5. Chakravarthy, U.S., Grant, J., Minker, J.: Logic-based approach to semantic query optimization. ACM Transactions on Database Systems 15(2), 162–207 (1990)
6. Codd, E.F.: Relational completeness of database subanguages. In: Database Systems, pp. 65–98 (1972)
7. Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM 25th Anniversary Issue 26(1), 64–69 (1983)
8. Cortesi, A., Zanioli, M.: Widening and narrowing operators for abstract interpretation. Computer Languages, Systems & Structures 37, 24–42 (2011)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the POPL 1977, pp. 238–252. ACM Press (1977)
10. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
11. Halder, R., Cortesi, A.: A Persistent Public Watermarking of Relational Databases. In: Jha, S., Mathuria, A. (eds.) ICISS 2010. LNCS, vol. 6503, pp. 216–230. Springer, Heidelberg (2010)
12. Halder, R., Cortesi, A.: Cooperative Query Answering by Abstract Interpretation. In: Černá, I., Gyimóthy, T., Hromkovič, J., Jefferey, K., Králović, R., Vukolić, M., Wolf, S. (eds.) SOFSEM 2011. LNCS, vol. 6543, pp. 284–296. Springer, Heidelberg (2011)
13. Halder, R.: Extending Abstract Interpretation to New Applicative Scenarios. Ph.D. thesis, Università Ca' Foscari Venezia (2012)

14. Halder, R., Cortesi, A.: Abstract interpretation of database query languages. Computer Languages, Systems & Structures 38, 123–157 (2012)
15. Halder, R., Cortesi, A.: Fine Grained Access Control for Relational Databases by Abstract Interpretation. In: Pedrosa, V. (ed.) ICSOFT 2010. CCIS, vol. 170, pp. 235–249. Springer, Heidelberg (2012)
16. Halder, R., Cortesi, A.: Abstract program slicing of database query languages. In: Proceedings of the the 28th Symposium On Applied Computing - Special Track on Database Theory, Technology, and Applications. ACM Press, Coimbra (2013)
17. Halder, R., Pal, S., Cortesi, A.: Watermarking techniques for relational databases: Survey, classification and comparison. Journal of Universal Computer Science 16(21), 3164–3190 (2010)
18. Logozzo, F.: Class invariants as abstract interpretation of trace semantics. Computer Languages, Systems & Structures 35, 100–142 (2009)
19. Ordonez, C.: Optimization of linear recursive queries in sql. IEEE Transactions on Knowledge and Data Engineering 22(2), 264–277 (2010)
20. Pieciukiewicz, T., Stencel, K., Subieta, K.: Usable Recursive Queries. In: Eder, J., Haav, H.-M., Kalja, A., Penjam, J. (eds.) ADBIS 2005. LNCS, vol. 3631, pp. 17–28. Springer, Heidelberg (2005)
21. Przymus, P., Boniewicz, A., Burzańska, M., Stencel, K.: Recursive Query Facilities in Relational Databases: A Survey. In: Zhang, Y., Cuzzocrea, A., Ma, J., Chung, K.-i., Arslan, T., Song, X. (eds.) DTA and BSBT 2010. CCIS, vol. 118, pp. 89–99. Springer, Heidelberg (2010)