

Abstract Interpretation for Sound Approximation of Database Query Languages

Raju Halder

Dipartimento di Informatica
Università Ca' Foscari di Venezia, Italy
halder@unive.it

Agostino Cortesi

Dipartimento di Informatica
Università Ca' Foscari di Venezia, Italy
cortesi@unive.it

Abstract—In this paper we extend the Abstract Interpretation framework to the field of query languages for relational databases as a way to support sound approximation techniques. This way, the semantics of query languages can be tuned according to suitable abstractions of the concrete domain of data.

Index Terms—Databases, Program Analysis, Abstract Interpretation

I. INTRODUCTION

In the context of web-based services interacting with DBMS, there is a need of “sound approximation” of database query languages, in order to minimize the weight of the database replicas in the web or in order to hide specific data values while giving them public access with larger granularity. There are many other application areas where processing of database information at different level of abstraction plays very important role, like, the applications where users are only interested in getting query answers based on some properties of data stored in the database rather than their actual values. The current query processing system requires the users to have precise information about the problem domain, database schema, and database content while it provides a limited answers and options, or even no information at all, if the exact information is not available. Cooperative query answering [1], [2], [3] supports query relaxation and provides intelligent, conceptual and approximate answers as well as exact answers where the data are organized into conceptual type abstraction hierarchies (TAH). Searching approximate values for a specialized value is equivalent to finding the abstract value of the specialized value, since the specialized values of the same abstract value constitute approximate values of one another. An example of cooperative query answering is that in response to the query about “specific flight departing at 10 a.m. from *Rome Fiumicino* airport to *Paris Orly* airport” the cooperative query processing may return “all flight information during morning time from airports in *Rome* to airports in *Paris*” and the user, thus, will be able to choose other flights if specific flight is unavailable. We may also mention another example from [2], for instance, in a personnel database example, suppose that a personnel manager wants to find out people

majoring in “finance” to fill-up certain vacant position. If the right candidate, majoring in “finance”, is unavailable or insufficient for a certain personnel management task, other employees with related major may be obtained by enlarging the scope of the search. The sound approximation of the database and its query languages may also serve as a formal foundation of answering queries approximately as a way to reduce query response times in on-line decision support systems, when the precise answer is not necessary or early feedback is helpful. Given the exploratory nature of such applications (*e.g.* decision support, experiment management, etc), many of these queries end up producing no result of particular interest to the user. Wasted time can be saved if users are able to quickly see an approximate answer to their query, and only proceed with the complete execution if the approximate answer indicated something interesting. However [1], [2], [3] do not provide a formal framework to prove the safety and soundness of the query processing.

We now mention another crucial application of safe and sound semantics-based approximation for database query languages. When a database is being populated with tuples, all tuples must satisfy some properties which are represented in terms of integrity constraints. For example, the age of the employees must be positive and must lie between 18 and 62. Any transaction over the database must satisfy all these integrity constraints as well. The dynamic checking for any transaction to ensure whether it violates the integrity constraints of the database can increase the run-time overhead significantly, while managing the integrity constraint verification statically may have a significant impact in terms of efficiency.

An interesting solution to all these problems can be provided by extending to the database field a well known static analysis technique, called Abstract Interpretation [4], [5], [6]. Abstract Interpretation, in fact, has been proved, in other contexts, as the best way to provide a semantics-based approach to approximation. Its main idea is to relate concrete and abstract semantics where the later are focussing only on some properties of interest. It was originally developed by Cousot and

Cousot as a unifying framework for designing and then validating static program analysis, and recently it becomes a general methodology for describing and formalizing approximate computation in many different areas of computer science, like model checking, verification of distributed memory systems, process calculi, security, type inference, constraint solving, etc.

Relational databases enjoy mathematical formulations that yield to a semantic description using formal language like relational algebra or relational calculus. To handle the aggregate functions or *NULL* values, some extensions of existing relational algebra and relational calculus have been introduced [7], [8], [9], [10]. However, this semantic description covers only a subset of *SQL* [7], [8], [11]. In particular, problems arise when dealing with *UPDATE*, *INSERT* or *DELETE* statements since operators originally proposed in relational algebra does not fully support them. This motivates our theoretical work aiming at defining a complete denotational semantics of *SQL* embedded applications, both at the concrete and at the abstract level, as a basis to develop an Abstract Interpretation of application programs embedded with *SQL*. The semantics is described by rules which specify how expressions are evaluated and commands are executed. As far as we know, the impact of abstract interpretation for sound approximation of database query languages has not yet been investigated. This is the aim of this paper.

The underlying concepts is that the applications embedded with *SQL* code basically interact with two worlds or environments: *user world* and *database world*. Corresponding to these two worlds or environments we define two sets of variables: \mathbb{V}_d and \mathbb{V}_a . The set \mathbb{V}_d is the set of database variables (*i.e.* the set of table attributes) and \mathbb{V}_a is a distinct set of variables called application variables defined in the application. Variables from \mathbb{V}_d are involved only in the *SQL* commands whereas variables in \mathbb{V}_a may occur in all type of instructions of the application. We denote any *SQL* command by a tuple $C_{sql} \triangleq \langle A_{sql}, \phi \rangle$. We call the first component A_{sql} the *active part* and the second component ϕ the *passive part* of C_{sql} . In an abstract sense, any *SQL* command C_{sql} 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_{sql} . The pre-condition ϕ appears in *SQL* commands as a well-formed formula in first-order logic. The semantics defined this way can be lifted from the concrete domain of values to abstract representation of them by providing suitable abstract operators corresponding to the concrete ones.

The structure of this paper is as follows: Section 2 recalls some preliminary concepts. Section 3 defines the abstract syntax of the application embedded with *SQL*. In Section 4 we define environments and states associated with an application. Section 5 describes the

semantics of the arithmetic and boolean expressions. Section 6 describes the formal semantics of atomic and composite statements. Section 7 describes the abstract semantics and discusses the practical impact of this technique. Section 8 concludes this paper.

II. PRELIMINARIES

In this section we recall some basic mathematical notation used in the literature, some ideas about Semantic Interpretation of First-Order Logic [12] and Abstract Interpretation [5].

If S and T are sets, then $\wp(S)$ denotes the powerset of S , $|S|$ the cardinality of S , $S \setminus T$ the set-difference between S and T , $S \times T$ the Cartesian product. A poset P with ordering relation \sqsubseteq is denoted as $\langle P, \sqsubseteq \rangle$, while $\langle C, \sqsubseteq, \sqcup, \sqcap, \top, \perp \rangle$ denotes the complete lattice C , with ordering \sqsubseteq , lub \sqcup , glb \sqcap , greatest element \top , and least element \perp .

We use the following functions in the subsequent section: $const(e)$ returns the constants involved in e ; $var(e)$ returns the variables involved in e ; $attr(t)$ returns the attributes associated with t ; $dom(f)$ returns the domain of f ; $target(f)$ returns a subset of $dom(f)$ on which the application of f is restricted.

Now we recall the concepts of semantic interpretation of the well-formed formula ϕ in first-order language L [12]. Let F , R and C be the sets of function symbols, relation symbols and constant symbols respectively in the first order language L . A semantic structure ζ for L is a non-empty set D_ζ , called the domain of the structure, along with: (i) for each function symbol $f_{n,m} \in F$, there is a function $f_{n,m}^\zeta : D_\zeta^n \rightarrow D_\zeta$; (ii) for each relation symbol $R_{n,m} \in R$, there is a subset $R_{n,m}^\zeta$ of D_ζ^n ; (iii) for each constant symbol $c_k \in C$, there is an element c_k^ζ . The subscript n in the notation $f_{n,m}$ and $R_{n,m}$ gives the number of arguments of the corresponding function or relation, whereas subscript m says it is the m^{th} of the symbols requiring n arguments. In general, a subset of D_ζ^n is called an n -place relation on D_ζ , so that the subsets $R_{n,m}^\zeta$ are just described as the relations on ζ , and the c_k^ζ s are called constants of ζ . The functions $f_{n,m}^\zeta \in F_\zeta$, relations $R_{n,m}^\zeta \in R_\zeta$ and constants $c_k^\zeta \in C_\zeta$ are called the interpretations in the structure ζ of the corresponding symbols of L .

We shall often write semantic structures using the notation, $\zeta = \langle D_\zeta, C_\zeta, F_\zeta, R_\zeta \rangle$. Let L be a language with equality. A structure for L is said to be normal if the interpretation of $=$ is equality on its domain.

Suppose that the variables x_1, x_2, \dots, x_n are interpreted respectively by elements a_1, a_2, \dots, a_n of D_ζ . We shall abbreviate this interpretation by \vec{a}/\vec{x} . Then the interpretation in ζ of each term $\tau \in \mathbb{T}$ of L under this interpretation of the variables, denoted by $\tau[\vec{a}/\vec{x}]^\zeta$, is defined recursively as follows:

- For each variable x_i , $x_i[\vec{a}/\vec{x}]^\zeta = a_i$.
- For each constant symbol c_k , $c_k[\vec{a}/\vec{x}]^\zeta = c_k^\zeta$.
- If $f_{n,m}$ is a function symbol in L and $\tau_1, \tau_2, \dots, \tau_n \in \mathbb{T}$, then $f_{n,m}(\tau_1, \tau_2, \dots, \tau_n)[\vec{a}/\vec{x}]^\zeta =$

$$f_{n,m}^c(\tau_1[\vec{a}/\vec{x}]^c, \dots, \tau_n[\vec{a}/\vec{x}]^c).$$

Now let ϕ be a well-formed-formula of L . The relation $\varsigma \models \phi[\vec{a}/\vec{x}]$ is read as "the formula ϕ is true in, or is satisfied by, the structure ς when x_1, x_2, \dots, x_n are interpreted by a_1, a_2, \dots, a_n ". This can be easily defined recursively on the construction of ϕ [12].

The basic idea of Abstract Interpretation [4], [5], [6] is that the program behavior at different levels of abstraction is an approximation of its formal concrete semantics. The abstract semantics is obtained from the concrete one by substituting concrete domains of computation and their basic concrete operations with abstract domains and corresponding abstract operations. The basic intuition is that abstract domains are representations of some properties of interest about concrete domains' values, while abstract operations simulate, over the properties encoded by the abstract domains, the behavior of their concrete counterparts.

The concrete semantics belongs to concrete semantics domain \mathcal{D}^c which is a complete lattice $\langle \mathcal{D}^c, \sqsubseteq^c \rangle$ partially ordered by \sqsubseteq^c . The ordering $A \sqsubseteq^c B$ implies that A is more precise(concrete) than B . The abstract semantics domain is also a complete lattice $\langle \mathcal{D}^a, \sqsubseteq^a \rangle$ ordered by abstract version \sqsubseteq^a of the concrete one \sqsubseteq^c .

The correspondence between these two concrete and abstract semantics domains \mathcal{D}^c and \mathcal{D}^a form a Galois connection $(\mathcal{D}^c, \alpha, \gamma, \mathcal{D}^a)$, denoted by $\mathcal{D}^c \xrightarrow{\alpha} \overleftarrow{\gamma} \mathcal{D}^a$, where the function $\alpha : \mathcal{D}^c \rightarrow \mathcal{D}^a$ and $\gamma : \mathcal{D}^a \rightarrow \mathcal{D}^c$ form an adjunction, namely $\forall A \in \mathcal{D}^a, \forall C \in \mathcal{D}^c : \alpha(C) \sqsubseteq^a A \Leftrightarrow C \sqsubseteq^c \gamma(A)$ where $\alpha(\gamma)$ is the left(right) adjoint of $\gamma(\alpha)$. α and γ are called abstraction and concretization maps respectively.

III. ABSTRACT SYNTAX

The abstract syntax of the application programs embedded with SQL is depicted in Table I. It is based on the following syntactic sets:

$n : \mathbb{Z}$	Integer
$k : \mathbb{S}$	String
$c : \mathbb{C}$	Constants
$v_a : \mathbb{V}_a$	Application Variables
$v_d : \mathbb{V}_d$	Database Variables (attributes)
$v : \mathbb{V} \triangleq \mathbb{V}_d \cup \mathbb{V}_a$	Variables
$e : \mathbb{E}$	Arithmetic Expressions
$b : \mathbb{B}$	Boolean Expressions
$A_{sql} : \mathbb{A}_{sql}$	SQL Actions
$\tau : \mathbb{T}$	Terms
$a_f : \mathbb{A}_f$	Atomic Formulas
$\phi : \mathbb{W}$	Well-formed formulas
$C_{sql} : \mathbb{C}_{sql}$	SQL Commands
$I : \mathbb{I}$	Instructions/Commands

The constant set is formed by the elements from the integer and string sets \mathbb{Z} and \mathbb{S} respectively. The passive part (*precondition*) ϕ of SQL command C_{sql} is a well-formed-formula in first order logic. We deal with only Data Manipulation Language (*DML*) for the active part A_{sql} of SQL command, that is, an SQL action is the

application of either *SELECT*, or *UPDATE*, or *INSERT*, or *DELETE*.

The function $\text{GROUP BY}(\vec{e})[t]$ where \vec{e} represents an ordered sequence of expressions, is applied on a table t and depending on the values of \vec{e} , it results into maximal partitions over the rows of t . The functions $\text{ORDER BY ASC}(\vec{e})[t]$ and $\text{ORDER BY DESC}(\vec{e})[t]$ sort the rows of the table t in ascending or descending order based on the value of \vec{e} . Observe that, the active part A_{sql} of *SELECT* statement may or may not use "GROUP BY" and "ORDER BY" functions and this fact is reflected by g and f respectively.

It should be noted that, if *SELECT* statement uses $\text{GROUP BY}(\vec{e})$, then there must be an $\vec{h}(\vec{x})$ which is applied on each partition obtained by GROUP BY operation. In that case, the i^{th} element $h_i \in \vec{h}$ must be *DISTINCT* function iff corresponding j^{th} element of \vec{x} i.e. x_j belongs to $\vec{x} \cap \vec{e}$. If x_j is $*$ then h_i must be *COUNT*. Otherwise, $h_i(x_i)$ is $s \circ r(e)$ where $e = x_i \wedge e \notin \vec{x} \cap \vec{e}$. That is,

$$h_i \triangleq \begin{cases} \text{DISTINCT} & \text{if } x_i \in \vec{x} \cap \vec{e} \text{ or,} \\ \text{COUNT} & \text{if } x_i = * \text{ or,} \\ s \circ r & \text{otherwise} \end{cases}$$

When the *SELECT* statement does not use any $\text{GROUP BY}(\vec{e})$ function, then $\vec{h}(\vec{x})$ is performed only over one group containing all tuples of the table. In that case, either each $h_i \in \vec{h}$ represents the identity function id or

$$h_i \triangleq \begin{cases} \text{COUNT} & \text{if } x_i = * \text{ or,} \\ s \circ r & \text{otherwise} \end{cases}$$

Note that, the function r involved in $h_i \in \vec{h}$ deals with duplicate values of the argument expression e of h_i , whereas the function r in $r(\vec{h}(\vec{x}))$ of the active part A_{sql} in *SELECT* statement deals with duplicate results obtained after performing \vec{h} over the group(s).

The ϕ in the active part A_{sql} of *SELECT* statement represents the "Having" clause which filters out only the valid groups obtained by grouping. The variable v_a is the Record/ResultSet type application variable with an ordered sequence of fields \vec{w} . The type of each field $w_i \in \vec{w}$ is the same as the return type of the corresponding function $h_i(x_i) \in \vec{h}(\vec{x})$. By the vector notation \vec{v}_d , we denote an ordered sequence of database variables. Finally, we introduce a particular assignment " $v_a := ?$ ", called random assignment, in the instruction set, that models the insertion of input values at run time by an external user.

IV. ENVIRONMENT AND STATE

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 the semantic domain \mathcal{D}_{\cup} where $\mathcal{D}_{\cup} = \{\mathcal{D} \cup \{\cup\}\}$, where \cup represents the undefined value.

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

$c ::=$	$n \mid k$
$e ::=$	$c \mid v_d \mid v_a \mid op \ e \mid e_1 \ op \ e_2$ where, op is an arithmetic operator.
$b ::=$	$e_1 = e_2 \mid e_1 \ op_r \ e_2 \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid true \mid false$ where op_r is an relational operator.
$\tau ::=$	$c \mid v_a \mid v_d \mid f_n(\tau_1, \tau_2, \dots, \tau_n)$ where, f_n is an n-ary function.
$a_f ::=$	$R_n(\tau_1, \tau_2, \dots, \tau_n) \mid \tau_1 = \tau_2$ where, R_n is an n-ary relation.
$\phi ::=$	$a_f \mid \neg \phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x_i \ \phi \mid \exists x_i \ \phi$
$g(\vec{e}) ::=$	$GROUP \ BY(\vec{e}) \mid id$
$r ::=$	$DISTINCT \mid ALL$
$s ::=$	$AVG \mid SUM \mid MAX \mid MIN \mid COUNT$
$h(e) ::=$	$s \circ r(e) \mid DISTINCT(e) \mid id$
$h(*) ::=$	$COUNT(*)$
$\vec{h}(\vec{x}) ::=$	$\langle h_1(x_1), \dots, h_n(x_n) \rangle$, where $\vec{h} = \langle h_1, \dots, h_n \rangle$ and $\vec{x} = \langle x_1, \dots, x_n \rangle$
$f(\vec{e}) ::=$	$ORDER \ BY \ ASC(\vec{e}) \mid ORDER \ BY \ DESC(\vec{e}) \mid id$
$A_{sql} ::=$	$select(v_a, f(\vec{e}), r(\vec{h}(\vec{x})), \phi, g(\vec{e})) \mid update(\vec{v}_d, \vec{e}) \mid insert(\vec{v}_d, \vec{e}) \mid delete$
$C_{sql} ::=$	$\langle A_{sql}, \phi \rangle \mid C'_{sql} \ UNION \ C''_{sql} \mid C'_{sql} \ INTERSECT \ C''_{sql} \mid C'_{sql} \ MINUS \ C''_{sql}$
$I ::=$	$skip \mid v_a := e \mid v_a := ? \mid C_{sql} \mid if \ b \ then \ I_1 \ else \ I_2 \mid while \ b \ do \ I \mid I_1; I_2$

TABLE I

ABSTRACT SYNTAX OF THE APPLICATION PROGRAM EMBEDDED WITH SQL

Definition 2: (Database Environment) A database is a set of tables $\{t_i \mid i \in I\}$ for a given set of indexes I . We may define a function ρ_d whose domain is I , such that for $i \in I$, $\rho_d(i) = t_i$.

In the Example depicted in Figure 1, the index set I is $\{emp, dept\}$, and the database d is the set $\{t_{emp}, t_{dept}\}$. So, $\rho_d(emp) = t_{emp}$.

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

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

That is ρ_t maps a_i to the ordered set of values over the rows of the table t where j ranges over the list of rows in t .

In the Example of Figure 1, $dom(\rho_{t_{emp}}) = \{eID, Name, Age, Dno, Pno, Sal, Child-no\}$ and, $\rho_{t_{emp}}(age) = \langle 30, 22, 50, 10, 40, 70, 18, 14 \rangle$.

Given a database d and a table $t_i \in d$ with $\vec{a} = attr(t_i)$. We can write, $\rho_d(i) = \rho_{t_i}(\vec{a})$. Given a SQL embedded program P , we define a state $\sigma \in \mathfrak{S}$ as a triplet $\langle I, \rho_d, \rho_a \rangle$ where I is the instruction to be executed, ρ_d and ρ_a are the database environment and application environment respectively on which I is executed. Thus, $\mathfrak{S} \triangleq \mathbb{I} \times \mathfrak{C}_d \times \mathfrak{C}_a$. The set of states of a program P is defined as $\mathfrak{S}[[P]] \triangleq P \times \mathfrak{C}[[P]]$ where, $\mathfrak{C}[[P]] \triangleq \mathfrak{C}_d \times \mathfrak{C}_a$. The

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

(a) t_{emp}

Deptno	Dname	Loc	MngRID
1	Math	Turin	4
2	Computer	Venice	1
3	Physics	Mestre	5

(b) t_{dept} Fig. 1. An example database d containing (a) t_{emp} , (b) t_{dept}

state transition relation is defined as $\Gamma \triangleq \mathfrak{S} \mapsto \wp(\mathfrak{S})$. The transitional semantics of a program P is thus defined as $\Gamma[[P]] \triangleq \mathfrak{S}[[P]] \mapsto \wp(\mathfrak{S}[[P]])$.

V. FORMAL SEMANTICS OF EXPRESSIONS

The evaluation of arithmetic expressions is defined by distinguishing different basic cases:

- 1) $E[[c]](\rho_d, \rho_a) = c$
- 2) $E[[v_a]](\rho_d, \rho_a) = \rho_a(v_a)$
- 3) $E[[v_d]](\rho_d, \rho_a)$
let $\exists t \in dom(\rho_d) : v_d = a_i \in attr(t)$
 $= E[[v_d]](\rho_t, \rho_a)$
 $= \rho_t(a_i)$
- 4) $E[[v_d \ op \ c]](\rho_d, \rho_a)$
let $\exists t \in dom(\rho_d) : v_d = a_i \in attr(t)$ and $op : D_i \times D_j \rightarrow D_k$ in
 $= E[[v_d \ op \ c]](\rho_t, \rho_a)$
 $= \langle (m \ op \ c) \in D_k \mid m \in \rho_t(a_i) \wedge a_i \in D_i \wedge c \in D_j \rangle$
- 5) $E[[v_d \ op \ v_a]](\rho_d, \rho_a)$
let $\exists t \in dom(\rho_d) : v_d = a_i \in attr(t)$ and $op : D_i \times D_j \rightarrow D_k$ in
 $= E[[v_d \ op \ v_a]](\rho_t, \rho_a)$
 $= \langle (m \ op \ n) \in D_k \mid m \in \rho_t(a_i) \wedge \rho_a(v_a) = n \wedge a_i \in D_i \wedge v_a \in D_j \rangle$
- 6) $E[[v_{d_1} \ op \ v_{d_2}]](\rho_d, \rho_a)$
let $\exists t \in dom(\rho_d) : v_{d_1} = a_i, v_{d_2} = a_j, \{a_i, a_j\} \subseteq attr(t)$ and
 $op : D_i \times D_j \rightarrow D_k$ in
 $= E[[v_{d_1} \ op \ v_{d_2}]](\rho_t, \rho_a)$
 $= \langle m_r \in D_k \mid m_r = \pi_i(l_r) \ op \ \pi_j(l_r) \text{ where } l_r \text{ is the } r^{th} \text{ row of } t \rangle$
- 7) $E[[e_1 \ op \ e_2]](\rho_d, \rho_a)$

$$= \begin{cases} \text{Case 1 :} \\ \exists! t \in dom(\rho_d) : \text{if } v_d \text{ occurs in } e_1 \text{ or } e_2 : v_d = a \in attr(t). \\ = E[[e_1 \ op \ e_2]](\rho_t, \rho_a) \\ = E[[e_1]](\rho_t, \rho_a) \ op \ E[[e_2]](\rho_t, \rho_a) \\ \\ \text{Case 2 :} \\ \text{Let } T = \{t \in dom(\rho_d) \mid \exists v_d \text{ occurring in } e_1 \text{ or } e_2 \text{ s.t.} \\ \quad v_d = a \in attr(t)\}. \\ \text{Let, } T = \{t_1, t_2, \dots, t_n\} \text{ and } t' = t_1 \times t_2 \times \dots \times t_n. \\ = E[[e_1 \ op \ e_2]](\rho_{t'}, \rho_a) \end{cases}$$

Finally, the evaluation of boolean expressions is defined as usual by structural induction, where $B[[true]](\rho_d, \rho_a) = true$ and $B[[false]](\rho_d, \rho_a) = false$.

VI. FORMAL SEMANTICS OF PROGRAM INSTRUCTIONS

The semantics $S[[I]](\rho_d, \rho_a)$ of a program instruction I in a SQL embedded program P defines the effect of executing this instruction on the environment ρ_a or (ρ_d, ρ_a) . There are two types of instructions: one executed only on ρ_a and other executed on both database and application environment (ρ_d, ρ_a) together. The SQL commands C_{sql} belong to the second category whereas, all other instructions of the application belong to the first category.

A. Semantics of SELECT

The Semantics of *SELECT* Statement C_{select} is described below:

$$S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})), \phi_1 \rangle]]_c(\rho_d, \rho_a) = \begin{cases} S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})), \phi_1 \rangle]]_c(\rho_t, \rho_a) \\ \text{if } \exists! t \in \text{dom}(\rho_d) : \\ \quad \text{target}(\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})), \phi_1 \rangle) = \{t\} \\ S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})), \phi_1 \rangle]]_c(\rho_{t'}, \rho_a) \text{ otherwise,} \\ \text{where } T = \{t_1, \dots, t_n \in \text{dom}(\rho_d) \mid t_i \text{ occurs in } C_{select}\} \text{ and} \\ t' = t_1 \times t_2 \times \dots \times t_n. \end{cases}$$

The semantics of *SELECT* is unfolded step by step:

Step 1. Absorbing ϕ_1 :

$$S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})), \phi_1 \rangle]]_c(\rho_{t'}, \rho_a) = S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})), true \rangle]]_c(\rho_{t'}, \rho_a) \text{ where,} \\ t' = \langle l_i \in t_0 \mid \text{let } \text{var}(\phi_1) = \vec{v}_a' \cup \vec{v}_a'' \text{ with } \vec{v}_a' = \vec{a} \subseteq \text{attr}(t_0) : \\ c \models \phi_1 [\pi_{\vec{a}}(l_i) / \vec{v}_a' \parallel \rho_a(\vec{v}_a'') / \vec{v}_a''] \rangle$$

Step 2. Grouping:

$$S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi, g(\vec{e})), true \rangle]]_c(\rho_t, \rho_a) = S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi, id), true \rangle]]_c(\cup_i \rho_{t_i}, \rho_a) = S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi, id), true \rangle]]_c(\rho_T, \rho_a) \\ \text{where, } g(\vec{e}) = \text{Group By}(\vec{e}) \text{ and } g(\vec{e})[t] \text{ is the maximal partition } T = \{t_1, t_2, \dots, t_n\} \text{ of } t, \text{ s.t. } \forall t_i \in T, t_i \subseteq t \\ \text{and } \forall e_i \in \vec{e}, \forall m_k, m_l \in E[[e_i]](\rho_{t_i}, \rho_a) : m_k = m_l$$

Step 3. Absorbing ϕ :

$$S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi, id), true \rangle]]_c(\rho_T, \rho_a) = S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), \phi, id), true \rangle]]_c(\cup_i \rho_{t_i}, \rho_a) \text{ where, } t_i \in T \\ = S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), true, id), true \rangle]]_c(\rho_T, \rho_a)$$

Where T' is defined as follows : there is a sequence of functions \vec{h}' occurring in ϕ , operating on groups, s.t. :

$$\vec{h}'(\vec{x}') \ni h'_i(x'_i) \triangleq \begin{cases} \text{DISTINCT}(e) \text{ or,} \\ s \circ r(e) \text{ or,} \\ \text{COUNT}(\ast) \end{cases}$$

Let \vec{v}_a' be a sequence of appl. variables occurring in ϕ

and, $\forall t_i \in T, \vec{h}'(\langle E[[\vec{x}]](\rho_{t_i}, \rho_a) \rangle) = \vec{c}_i$

and, $T' = \{t_i \in T \mid c \models \phi[\vec{c}_i / \vec{h}'(\vec{x}')][\rho_a(\vec{v}_a') / \vec{v}_a']\}$

Step 4. Applying $r(\vec{h}(\vec{x}))$ on each group:

$$= S[[\langle select(v_a, f(\vec{e}^t), r(\vec{h}(\vec{x})), true, id), true \rangle]]_c(\rho_T, \rho_a) = S[[\langle select(v_a, f(\vec{e}^t), id, true, id), true \rangle]]_c(\rho_{t'}, \rho_a)$$

where, $t' = \langle \vec{h}'(E[[\vec{x}]](\rho_{t_i}, \rho_a)) \mid t_i \in T \rangle$ and $t = r[t']$

Step 5. Possibly applying the ordering:

$$S[[\langle select(v_a, f(\vec{e}^t), id, true, id), true \rangle]]_c(\rho_{t'}, \rho_a) = S[[\langle select(v_a, id, id, true, id), true \rangle]]_c(\rho_{t'}, \rho_a) \text{ where, } t' = f(\vec{e})[t]$$

Step 6. Set the resulting values to the Record/ResultSet type application variable v_a with fields \vec{w} :

$$S[[\langle select(v_a, id, id, true, id), true \rangle]]_c(\rho_{t'}, \rho_a) = (\rho_{t_0}, \rho_{a'}) \text{ where, } \rho_{a'} = \rho_a[\rho_i(\vec{a}) / v_a(\vec{w})] \text{ with } \vec{a} = \text{attr}(t) \text{ and } t_0 \text{ is the initial table of step 1. Here, the } i^{\text{th}} \text{ field } w_i \in \vec{w} \text{ is substituted by the values of } i^{\text{th}} \text{ attribute } a_i \in \vec{a}.$$

Example 1: Consider the database of Figure 1 and the following statement:

```
SELECT DISTINCT Dno, Pno, MAX(Sal), AVG(DISTINCT Age), COUNT(*) FROM temp INTO  $\vec{v}_a$  WHERE Sal > 1000 GROUP BY (Dno, Pno) HAVING MAX(Sal) < 4000 ORDER BY AVG(DISTINCT Age), Dno
```

An equivalent formulation is:

```
SELECT DISTINCT ((DISTINCT(Dno), DISTINCT(Pno), MAX $\circ$  ALL(Sal), AVG $\circ$  DISTINCT(Age), COUNT(*))) FROM temp INTO  $\vec{v}_a$  WHERE Sal > 1000 GROUP BY ((Dno, Pno)) HAVING (MAX $\circ$  ALL(Sal) < 4000 ORDER BY (AVG(DISTINCT Age), Dno)
```

According to the abstract syntax, we get:

- $\phi_1 ::= \text{Sal} > 1000$
- $\vec{e} ::= \langle \text{Dno}, \text{Pno} \rangle$
- $g(\vec{e}) ::= \text{GROUP BY}(\langle \text{Dno}, \text{Pno} \rangle)$
- $\phi_2 ::= (\text{MAX} \circ \text{ALL}(\text{Sal})) < 4000$
- $\vec{h}' ::= \langle \text{DISTINCT}, \text{DISTINCT}, \text{MAX} \circ \text{ALL}, \text{AVG} \circ \text{DISTINCT}, \text{COUNT} \rangle$
 $\vec{x}' ::= \langle \text{Dno}, \text{Pno}, \text{Sal}, \text{Age}, \ast \rangle$
 $\vec{h}(\vec{x}') ::= \langle \text{DISTINCT}(\text{Dno}), \text{DISTINCT}(\text{Pno}), \text{MAX} \circ \text{ALL}(\text{Sal}), \text{AVG} \circ \text{DISTINCT}(\text{Age}), \text{COUNT}(\ast) \rangle$
- $\vec{e}^t ::= \langle \text{AVG}(\text{DISTINCT Age}), \text{Dno} \rangle$, where $\text{AVG}(\text{DISTINCT Age})$ simply represents an expression.
- $f(\vec{e}^t) ::= \text{ORDER BY ASC}(\langle \text{AVG}(\text{DISTINCT Age}), \text{Dno} \rangle)$
- $\vec{v}_a' ::= \text{Record or ResultSet type application variable with fields } \vec{w} = \langle w_1, w_2, w_3, w_4, w_5 \rangle$. The type of w_1, w_2, w_3, w_4, w_5 are same as the return type of $\text{DISTINCT}(\text{Dno}), \text{DISTINCT}(\text{Pno}), \text{MAX} \circ \text{ALL}(\text{Sal}), \text{AVG} \circ \text{DISTINCT}(\text{Age}), \text{COUNT}(\ast)$ respectively. For instance, in java as a host language, \vec{v}_a represents the object of the type *resultset*. Observe that, here we use the term "INTO" to understand the assignment to the application variable.

in:

```
SELECT  $r(\vec{h}(\vec{x}'))$  FROM temp INTO  $v_a(\vec{w})$  WHERE  $\phi_1$  GROUP BY  $\vec{e}$  HAVING  $\phi_2$  ORDER BY ASC  $\vec{e}^t$ 
```

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

(a) Absorbing "WHERE" Clause ϕ_1

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

(b) Grouping

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

(c) Absorbing "HAVING" Clause ϕ_2

Dno	Pno	MAX(Sal)	AVG(DISTINCT Age)	COUNT(*)
2	1	2000	30	1
1	2	1900	34	3
3	4	3000	40	1

(d) Performing $\vec{h}(\vec{x})$

Dno	Pno	MAX(Sal)	AVG(DISTINCT Age)	COUNT(*)
2	1	2000	30	1
1	2	1900	34	3
3	4	3000	40	1

(e) Getting table out of the result from (d)

Dno	Pno	MAX(Sal)	AVG(DISTINCT Age)	COUNT(*)
2	1	2000	30	1
1	2	1900	34	3
3	4	3000	40	1

(f) Elimination of duplicates

Dno	Pno	MAX(Sal)	AVG(DISTINCT Age)	COUNT(*)
2	1	2000	30	1
1	2	1900	34	3
3	4	3000	40	1

(g) Ordering

w_1	w_2	w_3	w_4	w_5
2	1	2000	30	1
1	2	1900	34	3
3	4	3000	40	1

(h) Assign to \vec{v}_a

TABLE II

OPERATIONS OF SELECT STATEMENT

We now illustrate the evaluation steps:

Step 1: Absorbtion:

Since $target(\langle select(\vec{v}_a, f(\vec{e}'), r(\vec{h}(\vec{x})), \phi_2, \vec{g}(\vec{e})), \phi_1 \rangle) = \{t_{emp}\}$, we apply the "WHERE" clause $\phi_1 ::= Sal > 1000$ on t_{emp} . The result is in Table II(a). The row "eID:7; Name:Alberto; Age:18; Dno:3; Pno:4; Sal:800; Child-no:1" is disregarded because, $\varsigma \not\models \phi_1[800/Sal]$. In fact, the semantic structure ς does not satisfy ϕ_1 when the variable 'Sal' is substituted by the value '800' of the corresponding row.

Step 2: Grouping:

By applying $g(\vec{e}') ::= GROUP BY(\langle Dno, Pno \rangle)$ on the

result of step 1 based on the argument $(\langle Dno, Pno \rangle)$, we get Table II(b). Here, we get 4 different groups with $(2,1), (1,2), (2,3)$ and $(3,4)$ as the values of $(\langle Dno, Pno \rangle)$ for the partitions t_1, t_2, t_3 and t_4 respectively.

Step 3: We apply the "HAVING" clause $\phi_2 ::= MAX \circ ALL(Sal) < 4000$ over all the groups in step 2. One group with the value of $(\langle Dno, Pno \rangle)$ equal to $(2,3)$ has been disregarded since the maximum salary of that group is not less than 4000. That is, the semantic structure ς does not satisfy ϕ_2 after interpreting $MAX \circ ALL(Sal)$ in ϕ_2 with the value return by the function $MAX \circ ALL(Sal)$ applied on that group. The result is shown in Table II(c).

Step 4: We perform $r(\vec{h}(\vec{x})) ::= DISTINCT(\langle DISTINCT(Dno), DISTINCT(Pno), MAX \circ ALL(Sal), AVG \circ DISTINCT(Age), COUNT(*) \rangle)$ on each group resulting from step 3.

(a) After applying $\vec{h}(\vec{x})$ on each group, we get the result as in Table II(d).

(b) Get the table t out of the results obtained in step (a): This is shown in Table II(e).

(c) Apply $r ::= DISTINCT$ over the rows of table t obtained in step (b): We get Table II(f) which is equal to Table II(e), since there are no duplicate rows.

Step 5: Performing $f(\vec{e}') ::= ORDER BY ASC(AVG(DISTINCT Age), Dno)$ on Table II(f) we get Table II(g).

Step 6: The result obtained in step 5 is assigned to the application variable \vec{v}_a . The result is shown in Table II(h).

B. Semantics of UPDATE

The update statement always targets an individual table. Let $target(\langle update(\vec{v}_a, \vec{e}), \phi \rangle) = \{t\}$, where $t \in dom(\rho_a)$.

$$\begin{aligned} S[\langle update(\vec{v}_a, \vec{e}), \phi \rangle]_{\varsigma}(\rho_a, \rho_a) \\ = S[\langle update(\vec{v}_a, \vec{e}), \phi \rangle]_{\varsigma}(\rho_t, \rho_a) \end{aligned}$$

Below, the semantics of UPDATE statement is unfolded step by step:

Step 1: Absorbing ϕ :

$$\begin{aligned} S[\langle update(\vec{v}_a, \vec{e}), \phi \rangle]_{\varsigma}(\rho_t, \rho_a) \\ = S[\langle update(\vec{v}_a, \vec{e}), true \rangle]_{\varsigma}(\rho_t, \rho_a) \end{aligned}$$

where,

$$\begin{aligned} t' = \langle l_i \in t \mid let \ var(\phi) = \vec{v}'_a \cup \vec{v}_a \text{ with } \vec{v}'_a = \\ \vec{a} \subseteq attr(t) : \varsigma \models \phi[\pi_a(l_i)/\vec{v}'_a \parallel \rho_a(\vec{v}_a)/\vec{v}_a] \rangle \end{aligned}$$

Step 2: Update:

$$S[\langle update(\vec{v}_a, \vec{e}), true \rangle]_{\varsigma}(\rho_t, \rho_a)$$

$$= (\rho_{t'}, \rho_a)$$

where, let $\vec{v}_d = \vec{a} \subseteq \text{attr}(t)$ and $\vec{e} = \langle e_1, e_2, \dots, e_h \rangle$
and $E[\vec{e}](\rho_t, \rho_a) = \langle \vec{m}_i \mid i = 1, \dots, h \rangle$.

let m_i^j be the j^{th} element of the sequence \vec{m}_i and
 a_i be the i^{th} element of the sequence \vec{a} .

$$t' = \langle l_j[m_i^j/a_i] \mid l_j \in t \rangle$$

C. Semantics of INSERT

Also the insert statement always targets an individual table. Let $t \in \text{dom}(\rho_d) : \text{target}(\langle \text{insert}(\vec{v}_d, \vec{e}), \Omega \rangle) = \{t\}$.

$$\begin{aligned} & S[\langle \text{insert}(\vec{v}_d, \vec{e}), \phi \rangle]_{\subseteq}(\rho_d, \rho_a) \\ &= S[\langle \text{insert}(\vec{v}_d, \vec{e}), \phi \rangle]_{\subseteq}(\rho_t, \rho_a) \\ &= S[\langle \text{insert}(\vec{v}_d, \vec{e}), \text{true} \rangle]_{\subseteq}(\rho_t, \rho_a) \\ &= (\rho_{t'}, \rho_a) \end{aligned}$$

where, $\vec{v}_d = \vec{a} \subseteq \text{attr}(t)$, $E[\vec{e}](\rho_a) = \vec{x}$, $\vec{a} = \langle a_1, a_2, \dots, a_n \rangle$, $\vec{x} = \langle x_1, x_2, \dots, x_n \rangle$, and $l_{\text{new}} = \langle x_1/a_1, x_2/a_2, \dots, x_n/a_n \rangle$, in $t' = t \cup \{l_{\text{new}}\}$

D. Semantics of DELETE

Let $t \in \text{dom}(\rho_d) : \text{target}(\langle \text{delete}, \phi \rangle) = \{t\}$, in

$$\begin{aligned} & S[\langle \text{delete}, \phi \rangle]_{\subseteq}(\rho_d, \rho_a) \\ &= S[\langle \text{delete}, \phi \rangle]_{\subseteq}(\rho_t, \rho_a) \\ &= (\rho_{t'}, \rho_a) \text{ where} \end{aligned}$$

$$\begin{aligned} t_d &= \langle l_i \in t \mid \text{let } \text{var}(\phi) = \vec{v}_d' \cup \vec{v}_a' \text{ with } \vec{v}_d' = \vec{a} \subseteq \text{attr}(t) : \\ & \quad \subseteq \vdash \phi \mid \pi_{\vec{a}}(l_i) / \vec{v}_d' \mid \rho_a(\vec{v}_a') / \vec{v}_a' \mid \rangle \\ t' &= t \setminus t_d \end{aligned}$$

E. Semantics of composite instructions

The inference rules for composite instructions are obtained by induction:

$$\begin{aligned} & \frac{S[\llbracket C_{sql_1} \rrbracket](\rho_d, \rho_a) = t_1 \quad S[\llbracket C_{sql_2} \rrbracket](\rho_d, \rho_a) = t_2}{S[\llbracket C_{sql_1} \text{ UNION } C_{sql_2} \rrbracket](\rho_d, \rho_a) = t_1 \cup t_2} \\ & \frac{S[\llbracket C_{sql_1} \rrbracket](\rho_d, \rho_a) = t_1 \quad S[\llbracket C_{sql_2} \rrbracket](\rho_d, \rho_a) = t_2}{S[\llbracket C_{sql_1} \text{ INTERSECT } C_{sql_2} \rrbracket](\rho_d, \rho_a) = t_1 \cap t_2} \\ & \frac{S[\llbracket C_{sql_1} \rrbracket](\rho_d, \rho_a) = t_1 \quad S[\llbracket C_{sql_2} \rrbracket](\rho_d, \rho_a) = t_2}{S[\llbracket C_{sql_1} \text{ MINUS } C_{sql_2} \rrbracket](\rho_d, \rho_a) = t_1 \setminus t_2} \\ & \frac{S[\llbracket A_1 \rrbracket](\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'}) \quad S[\llbracket A_2 \rrbracket](\rho_{d'}, \rho_{a'}) = (\rho_{d''}, \rho_{a''})}{S[\llbracket A_1; A_2 \rrbracket](\rho_d, \rho_a) = (\rho_{d''}, \rho_{a''})} \end{aligned}$$

Consider the auxiliary conditional statement *cond*:

$$\begin{aligned} & \text{cond}(B[\llbracket b \rrbracket], S[\llbracket A_1 \rrbracket], S[\llbracket A_2 \rrbracket])(\rho_d, \rho_a) = (\rho_{d'}, \rho_{a'}) \text{ where,} \\ & \text{either } B[\llbracket b \rrbracket](\rho_d, \rho_a) = \text{true and } S[\llbracket A_1 \rrbracket](\rho_d, \rho_a) \triangleq (\rho_{d'}, \rho_{a'}) \\ & \text{or } B[\llbracket b \rrbracket](\rho_d, \rho_a) = \text{false and } S[\llbracket A_2 \rrbracket](\rho_d, \rho_a) \triangleq (\rho_{d'}, \rho_{a'}) \end{aligned}$$

The semantics of "if b then A_1 else A_2 " statement is expressed using the conditional statement *cond* as follows:

$$\begin{aligned} & S[\llbracket \text{if } b \text{ then } A_1 \text{ else } A_2 \rrbracket](\rho_d, \rho_a) \\ &= \text{cond}(B[\llbracket b \rrbracket], S[\llbracket A_1 \rrbracket], S[\llbracket A_2 \rrbracket])(\rho_d, \rho_a) \end{aligned}$$

Since, "while b do A " \equiv "if b then (A ; while b do A) else skip", the semantics of the "while b do A " statement is expressed as follows:

$$\begin{aligned} & S[\llbracket \text{while } b \text{ do } A \rrbracket](\rho_d, \rho_a) \\ &= S[\llbracket \text{if } b \text{ then } (A; \text{while } b \text{ do } A) \text{ else skip} \rrbracket](\rho_d, \rho_a) \\ &= \text{FIX } F \text{ where, } F(g) = \text{cond}(B[\llbracket b \rrbracket], g \circ S[\llbracket A \rrbracket], \text{id})(\rho_d, \rho_a) \\ & \text{and FIX is a fix - point operator.} \end{aligned}$$

Definition 4: (Equivalence of Instructions) Let the environments (ρ_d, ρ_a) and $(\rho_{d'}, \rho_{a'})$ be denoted by ρ_x and $\rho_{x'}$ respectively. Two instructions I_1 and I_2 are said to be equivalent iff, $\{(\rho_x, \rho_{x'}) \mid S[\llbracket I_1 \rrbracket](\rho_x) = \rho_{x'}\} = \{(\rho_x, \rho_{x'}) \mid S[\llbracket I_2 \rrbracket](\rho_x) = \rho_{x'}\}$. In other words, $I_1 \equiv I_2$ iff I_1 and I_2 determine the same partial function on states.

VII. THE ABSTRACT SYNTAX AND SEMANTICS OF SQL

We are now in position to tune the semantics on different levels of abstraction of the concrete values. The semantics of SQL operations defined so far can be lifted to an abstract settings, where instead of working on a concrete database, queries are applied to an abstract database, in which some information are disregarded and concrete values are possibly represented by suitable abstractions. We denote by the apex $\#$, the syntactic elements of the abstract semantics. For each concrete element z , whenever we use the syntax $z^\#$, this means that there is a monotonic representation function γ from the abstract to the concrete domain such that $z \sqsubseteq \gamma(z^\#)$.

The syntax of SQL statement $C^\#$ and SQL action $A^\#$ over an abstract domain corresponding to the concrete SQL command C_{sql} and action A_{sql} represented as below:

$$\begin{aligned} C^\# & ::= \langle A^\#, \phi^\# \rangle \mid C_1^\# \text{ UNION } C_2^\# \mid C_1^\# \text{ INTERSECT } C_2^\# \\ & \quad \mid C_1^\# \text{ MINUS } C_2^\# \\ A^\# & ::= \text{select}^\#(\vec{v}_a^\#, f^\#(e^\#), r^\#(h^\#(x^\#)), \phi^\#, g^\#(e^\#)) \mid \text{update}^\#(\vec{v}_d^\#, e^\#) \\ & \quad \mid \text{insert}^\#(\vec{v}_d^\#, e^\#) \mid \text{delete}^\# \end{aligned}$$

Different abstract functions/operators involved in $A^\#$ are:

$$\begin{aligned} g^\# & ::= \text{GROUP BY}^\# \mid \text{id} \\ r^\# & ::= \text{DISTINCT}^\# \mid \text{ALL}^\# \\ s^\# & ::= \text{AVG}^\# \mid \text{SUM}^\# \mid \text{MAX}^\# \mid \text{MIN}^\# \mid \text{COUNT}^\# \\ h^\#(e^\#) & ::= s^\# \circ r^\#(e^\#) \mid \text{DISTINCT}^\#(e^\#) \mid \text{id} \\ h^\#(*) & ::= \text{COUNT}^\#(*) \\ f^\# & ::= \text{ORDER BY ASC}^\# \mid \text{ORDER BY DESC}^\# \mid \text{id} \end{aligned}$$

The abstract elements of the abstract pre-condition $\phi^\#$ which is an abstraction of the first-order formula, are defined as follows:

$$\begin{aligned} \tau^\# & ::= c^\# \mid \vec{v}_d^\# \mid f_n^\#(\tau_1^\#, \tau_2^\#, \dots, \tau_n^\#) \text{ where, } f_n^\# = n - \text{ary function.} \\ a_f^\# & ::= R_n^\#(\tau_1^\#, \tau_2^\#, \dots, \tau_n^\#) \mid \tau_1^\# = \tau_2^\# \\ & \quad R_n^\# \text{ is an } n - \text{ary relation: } R_n^\#(\tau_1^\#, \tau_2^\#, \dots, \tau_n^\#) \in \{\text{true, false, } \top\} \\ & \quad \text{where } \top \text{ means "either true or false".} \\ \phi^\# & ::= a_f^\# \mid \neg \phi_1^\# \mid \phi_1^\# \vee \phi_2^\# \mid \phi_1^\# \wedge \phi_2^\# \mid \forall x_i^\# \phi_1^\# \mid \exists x_i^\# \phi_1^\# \end{aligned}$$

Arithmetic expressions in abstract domain are defined as expected, whereas boolean expressions are evaluated

into a 3-valued logics, where \top means "either true or false".

$$c^\# ::= n^\# \mid k^\#$$

$$e^\# ::= c^\# \mid v_a^\# \mid v_a^\# \mid op^\# e^\# \mid e_1^\# op^\# e_2^\#$$

where, $op^\#$ represents abstract arithmetic operator.

$$b^\# ::= e_1^\# op_r^\# e_2^\# \mid \neg b^\# \mid b_1^\# \vee b_2^\# \mid b_1^\# \wedge b_2^\# \mid true \mid false \mid \top$$

where $op_r^\#$ represents an abstract relational operator.

Instructions in abstract domain are as follows:

$$I^\# ::= skip \mid v_a^\# := e^\# \mid v_a^\# :=? \mid C^\# \mid if\ b^\# \ then\ I_1^\# \ else\ I_2^\# \\ \mid while\ b^\# \ do\ I^\# \mid I_1^\#, I_2^\#$$

Definition 5: (Soundness and Completeness) Let γ be a representation function. The soundness and completeness conditions for an abstract functions/operators $f^\#$ in abstract domain with respect to the corresponding concrete function f are:

$$f^\# \text{ is sound if } \gamma \circ f^\# \sqsupseteq f \circ \gamma$$

$$f^\# \text{ is complete if } \gamma \circ f^\# = f \circ \gamma$$

Now we describe the correspondence between the concrete and abstract functions/operations involved in SQL queries as below:

- 1) The correspondence between g and $g^\#$:

$$g ::= GROUP\ BY \mid id \\ g^\# ::= GROUP\ BY^\# \mid id$$

The function g results into maximal partition over the rows of the table t based on arguments which is an ordered sequence of expressions \vec{e} . Let $t^\#$ represents the abstract table obtained from the concrete table t with corresponding abstract version of \vec{e} as $\vec{e}^\#$. Now if $g^\#$ is applied on $t^\#$ based on the argument $\vec{e}^\#$, then the number of partitions in abstract domain would be less as we are losing some information due to abstraction. Hence in abstract domain more tuples will be in the same group *i.e.* some partitions in concrete domain will be merged together in abstract domain. Hence we can write, $\gamma \circ g^\# \sqsupseteq g \circ \gamma$.

- 2) The correspondence between r and $r^\#$:

$$r ::= DISTINCT \mid ALL \\ r^\# ::= DISTINCT^\# \mid ALL^\#$$

The function r is used to deal with the duplication. Whenever the representation function is injective, r is complete *i.e.* $\gamma \circ r^\# = r \circ \gamma$.

- 3) The correspondence between s and $s^\#$:

$$s ::= AVG \mid SUM \mid MAX \mid MIN \mid COUNT \\ s^\# ::= AVG^\# \mid SUM^\# \mid MAX^\# \mid MIN^\# \mid COUNT^\#$$

For all these arithmetic operations s , the corresponding abstract operations $s^\#$ has to be provided, satisfying $s^\#(X^\#) \sqsupseteq lub\{s(X) \mid X \in \gamma(X^\#)\}$

- 4) The correspondence between f and $f^\#$:

$$f ::= ORDER\ BY\ ASC \mid ORDER\ BY\ DESC \mid id \\ f^\# ::= ORDER\ BY\ ASC^\# \mid ORDER\ BY\ DESC^\# \mid id$$

The *ORDER BY* function f applied over a set of rows of a table t and it sorts them based on a sequence of expressions \vec{e} in ascending or descending order. As the representation function γ is assumed to be monotonic, the function f is complete in the sense that $\gamma \circ f^\# = f \circ \gamma$

- 5) The correspondence between R_n and its abstract version $R_n^\#$ involved in ϕ and $\phi^\#$ respectively, should guarantee that, if $R_n^\#(\tau_1^\#, \dots, \tau_n^\#)$ is true, then $\forall \tau_1 \in \gamma(\tau_1^\#), \dots, \tau_n \in \gamma(\tau_n^\#) : R_n(\tau_1, \dots, \tau_n)$ is true and if $R_n^\#(\tau_1^\#, \dots, \tau_n^\#)$ is false, then $\forall \tau_1 \in \gamma(\tau_1^\#), \dots, \tau_n \in \gamma(\tau_n^\#) : R_n(\tau_1, \dots, \tau_n)$ is false. The values of $\phi^\#$ belong to the set $\{true, false, \top\}$ according to the actual three-valued propositional logics. For instance, if we consider the binary relation ' $<$ ' among integers and the abstract domain is the domain of intervals, then the abstract relation ' $<^\#$ ' corresponding to ' $<$ ' is defined as follows:

$$[l_i, h_i] <^\# [l_j, h_j] \triangleq \begin{cases} true & \text{if } h_i < l_j \\ false & \text{if } h_j \leq l_i \\ \top & \text{otherwise} \end{cases}$$

- 6) The correspondence between f_n and $f_n^\#$ involved in the terms of well-formed-formula:

Soundness (and completeness eventually) rely on the local correctness of the operations in the abstract domain. For example, consider an abstract domain for parity. The ' \times ' operation over the concrete domain is mapped to its abstract version as, $odd(\times^\#)odd = odd$, $even(\times^\#)odd = even$ and $even(\times^\#)even = even$. If the abstract domain represents the sign property, then the corresponding operation would be $-(\times^\#)- = +$, $+(\times^\#)- = -$ and $+(\times^\#)+ = +$.

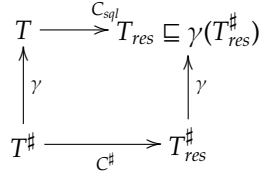
- 7) The correspondence between the instructions I and its abstract versions $I^\#$ is the expected one, except for the *conditional* and *while* statements:

- "if b then I_1 else I_2 " is abstracted by: "if $(b^\# = true)$ then $I_1^\#$ elseif $(b^\# = false)$ then $I_2^\#$ else $I_1^\# \sqcup I_2^\#$ ".

- "while b do I " is abstracted by $FIX\ F^\#$ where $F^\#$ is the functional corresponding to the concrete *while* statements.

Suppose, T and $T^\#$ represents a concrete and corresponding abstract tables respectively. The correspondence between T and $T^\#$ are described using the concretization and abstraction maps γ and α respectively. If C_{sql} and $C^\#$ are representing the SQL queries on concrete and abstract domains respectively, let T_{res} and $T_{res}^\#$ are the results of applying C_{sql} and $C^\#$ on the T and $T^\#$ respectively. The following fact illustrate

the soundness condition of abstraction:



Definition 6: (Soundness) Let $T^\#$ be an abstract table and $C^\#$ be an abstract query. $C^\#$ is sound iff $\forall T \in \gamma(T^\#), \forall C_{sql} \in \gamma(C^\#) : C_{sql}(T) \subseteq \gamma(C^\#(T^\#))$.

Example 2: Consider the database of Figure 1 which contains a concrete table t_{emp} and consider the following concrete *SELECT* statement:

```
SELECT Dno, MAX(Sal), MIN(Sal), AVG(Age), COUNT(*),
SUM(Child - no) FROM temp WHERE sal > 1600 GROUP BY
(Dno,Pno) HAVING MAX(sal) < 4000
```

If we execute the above query on Table 1(a), we get the following result:

Dno	MAX(Sal)	MIN(Sal)	AVG(Age)	COUNT(*)	SUM(Child - no)
2	2000	2000	30	1	4
3	3000	3000	40	1	5
1	1900	1700	40	2	3

Resulting table: t_{res} (concrete)

Table III shows the abstract table corresponding to Table 1(a) where the representation function α is defined as follows:

$$\begin{aligned}
 \alpha(\text{Age}) &\triangleq \begin{cases} [5,11] & \text{if } 5 \leq \text{age} \leq 11 \\ [12,24] & \text{if } 12 \leq \text{age} \leq 24 \\ [25, 59] & \text{if } 25 \leq \text{age} \leq 59 \\ [60, 100] & \text{if } 60 \leq \text{age} \leq 100 \end{cases} \\
 \alpha(\text{Sal}) &\triangleq \begin{cases} [500, 1499] & \text{if } 500 \leq \text{Sal} \leq 1499 \\ [1500,2499] & \text{if } 1500 \leq \text{Sal} \leq 2499 \\ [2500, 10000] & \text{if } 2500 \leq \text{Sal} \leq 10000 \end{cases} \\
 \alpha(\text{Child - no}) &\triangleq \begin{cases} \text{Few} & \text{if } 0 \leq \text{child - no} \leq 1 \\ \text{Medium} & \text{if } 2 \leq \text{child - no} \leq 3 \\ \text{many} & \text{iff } 4 \leq \text{child - no} \leq 10 \end{cases}
 \end{aligned}$$

The corresponding lattices for 'Age', 'Sal' and 'Child - no' in abstract domain are shown in figure 2, 3 and 4 respectively.

Let $S^\#$ be a set of abstract values or abstract elements

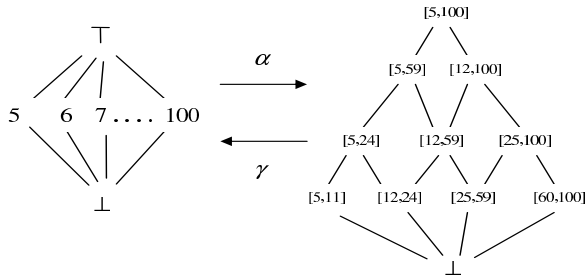


Fig. 2. Abstract lattice for attribute "Age"

i.e., $S^\# = \{ [l_i, h_i] \mid i \in I \subseteq \mathbb{N}; l_i, h_i \in \mathbb{Z}; l_i \leq h_i \}$. The abstract aggregate functions over $S^\#$ are defined as follows:

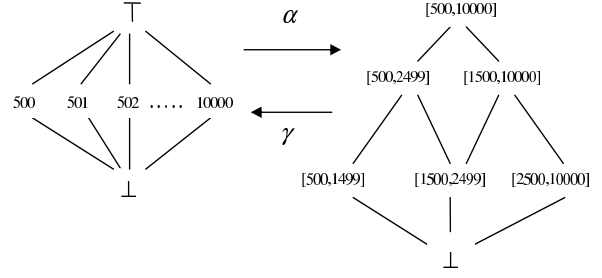


Fig. 3. Abstract lattice for attribute "Sal"

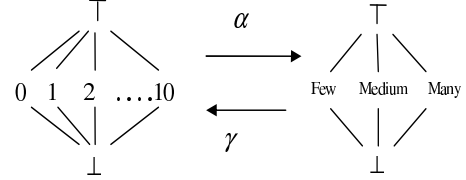


Fig. 4. Abstract lattice for attribute "Child - no"

$$AVG^\#(S^\#) \triangleq [\min_{i \in I}(l_i), \max_{i \in I}(r_i)]$$

$$SUM^\#(S^\#) \triangleq [\sum_{i \in I}(l_i), \sum_{i \in I}(r_i)]$$

$$MAX^\#(S^\#) \triangleq [\max_{i \in I}(l_i), \max_{i \in I}(r_i)]$$

$$MIN^\#(S^\#) \triangleq [\min_{i \in I}(l_i), \min_{i \in I}(r_i)]$$

The abstract version of $R_n^\#$ involved in pre-condition $\phi^\#$ are as follows:

$$[l_i, h_i] <^\# [l_j, h_j] \triangleq \begin{cases} \text{true} & \text{if } h_i < l_j \\ \text{false} & \text{if } h_j \leq l_i \\ \top & \text{otherwise} \end{cases}$$

$$[l_i, h_i] >^\# [l_j, h_j] \triangleq \begin{cases} \text{true} & \text{if } l_i > h_j \\ \text{false} & \text{if } l_j \geq h_i \\ \top & \text{otherwise} \end{cases}$$

The abstract *SELECT* statement becomes:

```
SELECT# Dno#, MAX#(sal#), MIN#(sal#), AVG#(age#), COUNT#(*),
SUM#(child - no#) FROM temp# WHERE sal# ># [1500,1499] GROUP
BY#(Dno#,Pno#) HAVING MAX#(sal#) <# [2500,10000]
```

The execution of the abstract query above is illustrated in Table IV:

Step 1: First apply the "WHERE" clause " $Sal^\# >^\# [1500, 2499]$ ". The result is shown in Table IV(a). Observe that soundness is preserved as we also include the rows where the evaluation of the relation $\geq^\#$ yields \top ; this might introduce inaccuracies, of course, in the abstract calculus, that results into a sound overapproximation of the concrete one.

Step 2: Apply, "GROUP BY[#](Dno[#],Pno[#])" and "HAVING" clause " $MAX^\#(Sal^\#) <^\# [2500,10000]$ ". The result is shown in Table IV(b).

Step 3: Apply, " $h^\#(x^\#) ::= \langle DISTINCT^\#(Dno^\#), MAX^\# \circ$

$eID^\#$	$Name^\#$	$Age^\#$	$Dno^\#$	$Pno^\#$	$Sal^\#$	$Child - no^\#$
1	Matteo	[25,59]	2	1	[1500,2499]	many
2	Alice	[12,24]	1	2	[1500,2499]	Medium
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]	Medium
7	Alberto	[12,24]	3	4	[500,1499]	Few
8	Bob	[12,24]	2	3	[2500,10000]	Medium

TABLE III
ABSTRACT TABLE: $t_{emp}^\#$

$eID^\#$	$Name^\#$	$Age^\#$	$Dno^\#$	$Pno^\#$	$Sal^\#$	$Child - no^\#$
1	Matteo	[25,59]	2	1	[1500,2499]	many
2	Alice	[12,24]	1	2	[1500,2499]	Medium
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]	Medium
7	Alberto	[12,24]	3	4	[500,1499]	Few
8	Bob	[12,24]	2	3	[2500,10000]	Medium

(a) Absorbing "WHERE" Clause $\phi_1^\#$

$eID^\#$	$Name^\#$	$Age^\#$	$Dno^\#$	$Pno^\#$	$Sal^\#$	$Child - no^\#$
1	Matteo	[25,59]	2	1	[1500,2499]	many
5	Deba	[25,59]	3	4	[2500,10000]	many
2	Alice	[12,24]	1	2	[1500,2499]	Medium
4	luca	[5,11]	1	2	[1500,2499]	Few
6	Andrea	[60,100]	1	2	[1500,2499]	Medium
3	Joy	[25,59]	2	3	[1500,2499]	Medium
8	Bob	[12,24]	2	3	[2500,10000]	Medium

(b) Grouping and Absorbing "HAVING" Clause $\phi_2^\#$

$Dno^\#$	$MAX^\#(Sal^\#)$	$MIN^\#(Sal^\#)$	$AVG^\#(Age^\#)$	$COUNT^\#(*)$	$SUM^\#(Child - no^\#)$
2	[1500,2499]	[1500,2499]	[25,59]	1	many
3	[2500,10000]	[2500,10000]	[25,59]	1	many
1	[1500,2499]	[1500,2499]	[5,100]	3	T
2	[2500,10000]	[1500,2499]	[12,59]	2	T

(c) Performing $\tilde{h}^\#(\tilde{x}^\#)$

$Dno^\#$	$MAX^\#(Sal^\#)$	$MIN^\#(Sal^\#)$	$AVG^\#(Age^\#)$	$COUNT^\#(*)$	$SUM^\#(Child - no^\#)$
2	[1500,2499]	[1500,2499]	[25,59]	1	many
3	[2500,10000]	[2500,10000]	[25,59]	1	many
1	[1500,2499]	[1500,2499]	[5,100]	3	T
2	[2500,10000]	[1500,2499]	[12,59]	2	T

(d) Resulting abstract table: $t_{res}^\#$

TABLE IV

ABSTRACT COMPUTATION OF $SELECT^\#$

$ALL^\#(Sal^\#)$, $MIN^\# \circ ALL^\#(Sal^\#)$, $AVG^\# \circ ALL^\#(Age^\#)$, $COUNT^\#(*)$, $SUM^\# \circ ALL^\#(Child - no^\#)$ ". We get the result in IV(c). Observe that $SUM^\#(\{a^\#\}) = a^\#$ and $SUM^\#(\{a^\#, b^\#\}) = \top$ if $a^\# \neq b^\#$.

Step 4: Finally, the abstract result $t_{res}^\#$ is shown in Table IV(d). Observe that the abstraction is sound i.e. $t_{res} \in \gamma(t_{res}^\#)$.

In the example discussed above we see how to design database abstraction, obtained by clustering numerical values into categorical ones. This approach can easily be applied in the web-services scenario in order to provide users either partial views or "customized replicas" of the database, where only the abstract values of the database are let available for downloading.

VIII. CONCLUSIONS

As far as we know this is the first attempt to formalize a Concrete/Abstract semantics for SQL query

languages within the Abstract Interpretation framework. This framework can serve several purposes like (i) Property-based query answering, (ii) Cooperative query processing, (ii) Static analysis for transaction to optimize the integrity constraint checking, (iii) To provide users either partial view or "customized replicas" of the database, etc. We are currently investigating its specialization for the analysis of security properties, e.g. for preventing SQL Injection attacks [13], or information leakage [14].

Acknowledgement

Work partially supported by Italian MIUR COFIN'07 project "SOFT".

REFERENCES

- [1] W. W. Chu, M. Merzbacher, and L. Berkovich, "The design and implementation of cobase," *SIGMOD Record*, vol. 22, no. 2, pp. 517–522, 1993.
- [2] S.-Y. Huh and K.-H. Moon, "Approximate query answering approach based on data abstraction and fuzzy relation," in *Proceedings of INFORMS-KORMS*. Seoul, Korea: The Korean Operations Research and Management Science Society, June 2000, pp. 2057–2065.
- [3] W. W. Chu and Q. Chen, "A structured approach for cooperative query answering," *IEEE Trans. on Knowl. and Data Eng.*, vol. 6, no. 5, pp. 738–749, 1994.
- [4] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, CA, USA: ACM Press, 1977, pp. 238–252.
- [5] R. Giacobazzi, F. Ranzato, and F. Scozzari, "Making abstract interpretations complete," *Journal of the ACM (JACM)*, vol. 47, no. 2, pp. 361–416, March 2000.
- [6] P. Cousot and R. Cousot, "Systematic design of program transformation frameworks by abstract interpretation," in *Conference Record of the 29th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*. Portland, OR USA: ACM Press, January 2002, pp. 178–190.
- [7] M. Negri, G. Pelagatti, and L. Sbatella, "Formal semantics of sql queries," *ACM Transactions on DataBase System*, vol. 17, no. 3, pp. 513–534, September 1991.
- [8] G. V. Bultzingwloewen, "Translating and optimizing sql queries having aggregates," in *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*. Brighton, England: Morgan Kaufmann Publishers Inc., September 1987, pp. 235–243.
- [9] A. Klug, "Equivalence of relational algebra and relational calculus query languages having aggregate functions," *Journal of the ACM (JACM)*, vol. 29, no. 3, pp. 699–717, July 1982.
- [10] R. Nakano, "Translation with optimization from relational calculus to relational algebra having aggregate functions," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 4, pp. 518–557, December 1990.
- [11] S. Ceri and G. Gottlob, "Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 324–345, 1985.
- [12] D. Goldrei, *Propositional and Predicate Calculus: A Model of Argument*. Springer, 2005.
- [13] W. G. Halfond, J. Viegas, and A. Orso, "A classification of sql injection attacks and countermeasures," in *Proceedings of the International Symposium on Secure Software Engineering (ISSSE 2006)*, Arlington, VA, USA, March 2006.
- [14] I. Mastroeni and R. Rossato, "Weakening non-interference on databases (abstract)," in *Workshop on Current and Emerging Research Issues in Computer Security, CERICS'06*, Royal Holloway, University of London, July 20-21 2006.