Computer Languages, Systems & Structures ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Abstract interpretation of database query languages

Raju Halder, Agostino Cortesi *

Università Ca' Foscari Venezia, Italy

ARTICLE INFO

Article history:
Received 3 March 2011
Received in revised form
5 July 2011
Accepted 28 October 2011

Keywords:
Databases
SQL
Program analysis
Abstract Interpretation

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. The abstraction of relational database system has many interesting applications, in particular, for security purposes, such as fine grained access control, watermarking, etc.

1. 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 database replicas on the web or in order to hide specific data values while giving them public access with larger granularity. There are many application areas where processing of database information at different level of abstraction plays important roles, like the applications where users are interested only in the query answers based on some properties of the database information rather than their exact values.

Given an exploratory nature of the applications, like decision support system, experiment management system, etc., many of the 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 indicates something interesting. 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, when the precise answer is not necessary or early feedback is helpful.

Cooperative query answering [1,2] supports query relaxation and provides intelligent, approximate answers as well as exact answers. It provides neighborhood or generalized information relevant to the original query and within a certain semantic distance of the exact answer. Searching approximate values for a specialized value is equivalent to find an abstract value of the specialized value, since the specialized values of the same abstract value constitute approximate values of one another. Sound approximation of the database system provides a formal framework to the field of cooperative query answering, ensuring three key issues: soundness, relevancy and optimality which are crucial in this context.

When a database is being populated with tuples, all tuples must satisfy some properties which are represented in terms of integrity constraints. For instance, the ages of the employees must be positive and must lie between 18 and 62. Any

* Corresponding author.
E-mail addresses: halder@unive.it (R. Halder), cortesi@unive.it, cortesi@dsi.unive.it (A. Cortesi).

1477-8424/$ - see front matter © 2011 Published by Elsevier Ltd.
doi:10.1016/j.cl.2011.10.004

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.

The traditional Fine Grained Access Control (FGAC) [3] provides only two extreme views to the database information: either public or private. There are many application areas where some partial or relaxed view of the confidential information is desirable. For instance, consider a database in an online transaction system containing the credit card numbers for its customers. According to the disclosure policy, the employees of the customer-care section are able to see the last four digits of the credit card numbers, whereas all the other digits are completely hidden. The traditional FGAC policy is unable to implement this type of security framework without changing the database structure.

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 [4–7]. 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. [7].

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 [8–11]. However, this semantic description covers only a subset of SQL [8,11,12]. In particular, problems arise when dealing with UPDATE, INSERT or DELETE statements since operators originally proposed in relational algebra do 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. In this setting, we represent all the syntactic elements in SQL statements (for example, GROUP BY, ORDER BY, DISTINCT clauses, etc.) as functions and the semantics is described as a partial functions on states which specify how expressions are evaluated and commands are executed. The functional representation of syntactic elements increases the power of expressibility of the semantics and facilitates us to provide a complete functional control on the corresponding domains of data. 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 database 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 *action part* and the second component ϕ the *pre-condition 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 C_{sql} 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[1] is as follows: Section 2 recalls some preliminary concepts. Section 3 defines the abstract syntax of the SQL embedded application. In Section 4, we define environments and states associated with the application. Section 5 describes the semantics of the arithmetic and boolean expressions, whereas Sections 6 and 7 describe the formal semantics of atomic and composite statements respectively. The correspondence of the proposed denotational semantic approach with the relational algebra is discussed in Section 8. In Section 9, we lift the syntax and semantics of the query languages from concrete domain to an abstract domain of interest by discussing the soundness and completeness of the abstraction in details. In Section 10, we discuss the formal semantics of SQL statements with correlated and non-correlated subquery. The interesting applications of suitable abstraction of the relational databases are discussed in Section 11. Section 12 discusses the related work in the literature. Finally, in Section 13, we draw our conclusions.

2. Preliminaries

In this section, we recall some basic mathematical notation used in the literature, some ideas about Semantic Interpretation of First-Order Logic [14] and the Abstract Interpretation theory [4–7].

2.1. Basic mathematical notation

If S and T are sets, then $\wp(S)$ denotes the powerset of S, $|S|$ the cardinality of S, $S \backslash T$ 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 $\langle C, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ denotes the complete lattice C with ordering $\sqsubseteq$, lub $\sqcup$, glb $\sqcap$, greatest element $\top$, and least element $\bot$.

[1] The paper is a revised and extended version of [13].

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

We use the following functions in the subsequent section:

- *const(e)* returns the constants appearing in *e*.
- *var(e)* returns the variables appearing 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.

2.2. Semantic interpretation of well-formed formulas in first-order language

We now recall the concepts of the semantic interpretation of a well-formed formula ϕ in a first-order language L [14].

Let F, R and C be the set of function symbols, relation symbols and constant symbols respectively in the first order language L. A semantic structure ς for L is a non-empty set D_ς, called the domain of the structure, along with the following:

1. For each function symbol $f_{n,m} \in F$, there is a function $f^\varsigma_{n,m} : D^n_\varsigma \to D_\varsigma$.
2. For each relation symbol $R_{n,m} \in R$, there is a subset $R^\varsigma_{n,m}$ of D^n_ς.
3. For each constant symbol $c_k \in C$, there is an element c^ς_k.

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 mth of the symbols requiring n arguments. In general, a subset of D^n_ς is called an n-place relation on D_ς, so that the subsets $R^\varsigma_{n,m}$ are just described as the relations on ς, and the c^ς_k s are called constants of ς. The functions $f^\varsigma_{n,m} \in F_\varsigma$, relations $R^\varsigma_{n,m} \in R_\varsigma$ and constants $c^\varsigma_k \in C_\varsigma$ are called the interpretations in the structure ς of the corresponding symbols of L.

We shall often write semantic structures using the notation, $\varsigma = \{D_\varsigma, C_\varsigma, F_\varsigma, R_\varsigma\}$. 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 $x_1, x_2, \ldots, x_n$ are interpreted respectively by elements $a_1, a_2, \ldots, a_n$ of D_ς. We shall abbreviate this interpretation by $\vec{a}/\vec{x}$. Then the interpretation in ς of each term $\tau \in \mathbb{T}$ of the first-order language L under this interpretation of the variables, which we write as $\tau[\vec{a}/\vec{x}]^\varsigma$, is defined recursively as follows:

- For each variable x_i, we define $x_i[\vec{a}/\vec{x}]^\varsigma = a_i$.
- For each constant symbol c_k, we define $c_k[\vec{a}/\vec{x}]^\varsigma = c^\varsigma_k$.
- If $f_{n,m}$ is a function symbol in first-order language L and $\tau_1, \tau_2, \ldots, \tau_n \in \mathbb{T}$, then $f_{n,m}(\tau_1, \tau_2, \ldots, \tau_n)[\vec{a}/\vec{x}]^\varsigma = f^\varsigma_{n,m}(\tau_1[\vec{a}/\vec{x}]^\varsigma, \ldots, \tau_n[\vec{a}/\vec{x}]^\varsigma)$.

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

- Atomic formulas:

(a) for each relation symbol $R_{n,m}$ in L and terms $\tau_1, \tau_2, \ldots, \tau_n$: $\varsigma \vDash R_{n,m}(\tau_1, \tau_2, \ldots, \tau_n)[\vec{a}/\vec{x}]$ if and only if $(\tau_1[\vec{a}/\vec{x}]^\varsigma, \ldots, \tau_n[\vec{a}/\vec{x}]^\varsigma) \in R^\varsigma_{n,m}$;

(b) if $\tau_1 = \tau_2$ are terms, then $\varsigma \vDash (\tau_1 = \tau_2)[\vec{a}/\vec{x}]$ if and only if $\tau_1[\vec{a}/\vec{x}]^\varsigma = \tau_2[\vec{a}/\vec{x}]^\varsigma$.

- For any formula of one of the forms $\neg\phi$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\forall x_i\ \phi$, $\exists x_i\ \phi$ truth tables laws are followed, e.g.,

(a) $\varsigma \vDash (\neg\phi)[\vec{a}/\vec{x}]$ if and only if it is not the case that $\varsigma \vDash \phi[\vec{a}/\vec{x}]$.

(b) $\varsigma \vDash (\phi_1 \vee \phi_2)[\vec{a}/\vec{x}]$ if and only if $\varsigma \vDash \phi_1[\vec{a}/\vec{x}]$ or $\varsigma \vDash \phi_2[\vec{a}/\vec{x}]$.

(c) $\varsigma \vDash (\phi_1 \wedge \phi_2)[\vec{a}/\vec{x}]$ if and only if $\varsigma \vDash \phi_1[\vec{a}/\vec{x}]$ and $\varsigma \vDash \phi_2[\vec{a}/\vec{x}]$.

(d) $\varsigma \vDash (\forall x_i\ \phi)[\vec{a}/\vec{x}]$ if and only if for all $b \in D_\varsigma$, $\varsigma \vDash \phi[\vec{a}/\vec{x}[b/x_i]]$.

(e) $\varsigma \vDash (\exists x_i\ \phi)[\vec{a}/\vec{x}]$ if and only if there is some $b \in D_\varsigma$, $\varsigma \vDash \phi[\vec{a}/\vec{x}[b/x_i]]$.

2.3. Abstract interpretation

The basic idea of abstract interpretation [4–7] is that the program behavior at different levels of abstraction is an approximation of its formal concrete semantics. Approximated/abstract semantics is obtained from the concrete one by substituting concrete domains of computation and their basic concrete semantic operations with abstract domains and corresponding abstract semantics 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 abstract domains, the behavior of their concrete counterparts. Abstract interpretation formalizes the correspondence between the concrete semantics $S^c[\![P]\!]$ of syntactically correct program $P \in \mathbb{P}$ in a given programming language $\mathbb{P}$ and its abstract semantics $S^a[\![P]\!]$ which is a safe approximation of the concrete semantics $S^c[\![P]\!]$.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

The concrete semantics belongs to concrete semantics domain $\mathfrak{D}^c$ which is a complete lattice $\langle \mathfrak{D}^c, \sqsubseteq^c \rangle$ 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 $\langle \mathfrak{D}^a, \sqsubseteq^a \rangle$ ordered by abstract version $\sqsubseteq^a$ of the concrete one $\sqsubseteq^c$.

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

Let (C, α, γ, A) be a Galois connection, $f : C \to C$ be a concrete function and $f^\sharp : A \to A$ be an abstract function. $f^\sharp$ is a sound, *i.e.*, correct approximation of f if $f \circ \gamma \sqsubseteq \gamma \circ f^\sharp$. When the soundness condition is strengthened to equality, *i.e.*, when $f \circ \gamma = \gamma \circ f^\sharp$, the abstract function $f^\sharp$ is a complete approximation of f in A. This means that no loss of precision is accumulated in the abstract computation through $f^\sharp$. Let *uco* be the upper closure operator on a lattice. Given $A \in uco(C)$ and a semantic function $f : C \to C$, the notation $f^A \triangleq \alpha \circ f \circ \gamma$ denotes the best correct approximation of f in A. It has been proved that, given an abstraction A, there exists a complete approximation of $f : C \to C$ in A if and only if the best correct approximation f^A is complete [7]. This means that completeness of f^A is an abstract domain property, namely that it depends on the structure of the abstract domains only.

3. Abstract syntax

The abstract syntax of the application programs embedded with SQL is depicted in Table 1. 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) involved in SQL commands
$v : \mathbb{V} \triangleq \mathbb{V}_d \cup \mathbb{V}_a$	Variables
$e : \mathbb{E}$	Arithmetic expressions
$b : \mathbb{B}$	Boolean expressions
$A_{sql} : \mathbb{A}_{sql}$	Action part of SQL commands
$\tau : \mathbb{T}$	Terms
$a_f : \mathbb{A}_f$	Atomic formulas
$\phi : \mathbb{W}$	Well-formed formulas (pre-condition part of SQL commands)
$C_{sql} : \mathbb{C}_{sql}$	SQL commands
$I : \mathbb{I}$	Instructions/commands

Any constant $c \in \mathbb{C}$ appearing in SQL command C_{sql} is either an integer $n \in \mathbb{Z}$ or a string $k \in \mathbb{S}$. The *pre-condition* ϕ of C_{sql} is a well-formed formula in first order logic. We deal with only Data Manipulation Language (DML) for the *action part* A_{sql}, that is, an SQL action is the application of either SELECT, or UPDATE, or INSERT, or DELETE. Observe that the database variables from the set $\mathbb{V}_d$ can appear in C_{sql} only. Since the variables from $\mathbb{V}_d$ represent the attributes of the database tables, we assume that no two tables have the same attributes.

The function GROUP BY $(\vec{e})[t]$ where $\vec{e}$ represents an ordered sequence of arithmetic expressions, is applied on a table t and depending on the values of $\vec{e}$ over the tuples of t, it results into maximal partition of the tuples of t. The functions ORDER BY ASC $(\vec{e})[t]$ and ORDER BY DESC $(\vec{e})[t]$ sort the tuples of table t in ascending or descending order based on the value of $\vec{e}$ over the tuples in t respectively. Observe that, A_{sql} of SELECT statement may or may not use GROUP BY and ORDER BY functions, and this fact is reflected in the abstract syntax of g and f respectively in Table 1.

Table 1
Abstract syntax of the application program embedded with SQL.

$c ::=$	$n \mid k$
$e ::=$	$c \mid v_d \mid v_a \mid op_u\ e \mid e_1\ op_b\ e_2$, where op_u and op_b represent unary and binary arithmetic operators respectively.
$b ::=$	$e_1 = e_2 \mid e_1 \geq e_2 \mid e_1 \leq e_2 \mid e_1 > e_2 \mid e_1 < e_2 \mid e_1 \neq e_2 \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid true \mid false$
$\tau ::=$	$c \mid v_a \mid v_d \mid f_n(\tau_1, \tau_2, \ldots, \tau_n)$, where f_n is an n-ary function.
$a_f ::=$	$R_n(\tau_1, \tau_2, \ldots, \tau_n) \mid \tau_1 = \tau_2$, where R_n is an n-ary relation: $R_n(\tau_1, \tau_2, \ldots, \tau_n) \in \{true, false\}$
$\phi ::=$	$a_f \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall x_i\ \phi \mid \exists x_i\ \phi$
$g(\vec{e}) ::=$	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 $(*)$
$\vec{h}(\vec{x}) ::=$	$\langle h_1(x_1), \ldots, h_n(x_n) \rangle$, where $\vec{h} = \langle h_1, \ldots, h_n \rangle$ and $\vec{x} = \langle x_1, \ldots, x_n \rangle$
$f(\vec{e}) ::=$	ORDER BY ASC$(\vec{e}) \mid$ ORDER BY DESC$(\vec{e}) \mid id$
$A_{sql} ::=$	$select(v_a, f(\vec{e}), r(\vec{h}(\vec{x})), \phi, g(\vec{e})) \mid update(\vec{v_d}, \vec{e}) \mid insert(\vec{v_d}, \vec{e}) \mid delete(\vec{v_d})$
$C_{sql} ::=$	$\langle A_{sql}, \phi \rangle \mid C_{sql}'$ 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* $I \mid I_1; I_2$

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

The aggregate functions in SELECT statement are represented by s. The clauses DISTINCT and ALL are used to deal with duplicate values. We denote DISTINCT and ALL by the function r. By $\vec{h}(\vec{x})$, we denote an ordered sequence of functions operating on an ordered sequence of arguments $\vec{x}$, *i.e.*, each function $h_i \in \vec{h}$ operates on the corresponding argument $x_i \in \vec{x}$. The argument x_i is an expression e or a sequence of all attributes of the table denoted by $*$ in SQL.

It should be noted that, if SELECT statement uses GROUP BY ($\vec{e}$), then there must be an $\vec{h}(\vec{x})$ which is evaluated on each partition obtained by GROUP BY operation, yielding to a single tuple. In such case, the ith element $h_i \in \vec{h}$ must be DISTINCT function if the corresponding ith element of $\vec{x}$ (*i.e.* x_i) belongs to $\vec{x} \cap \vec{e}$, or h_i must be COUNT if x_i is $*$, otherwise $h_i(x_i)$ is $s \circ r(e)$ where $e \in \vec{x} \wedge e \notin (\vec{x} \cap \vec{e})$. That is,

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

When the SELECT statement does not use any GROUP BY ($\vec{e}$) function, we have two situations: (i) if $\vec{h} \neq \vec{id}$, then the set of all tuples in the table for which ϕ satisfies is considered as a single group and $\vec{h}(\vec{x})$ is evaluated on that group. In that case, $h_i \in \vec{h}$ is defined as follows:

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

(ii) if $\vec{h} = \vec{id}$, then each tuple in the table for which ϕ satisfies is considered as an individual group and $\vec{h}(\vec{x}) = \vec{x}$ is evaluated on each of these groups.

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

The formula ϕ and the variable v_a appearing in A_{sql} of the SELECT statement represent the HAVING clause and a Record/ResultSet type application variable respectively. v_a has an ordered sequence of fields $\vec{w}$ where 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 "$v_a \coloneqq ?$", called random assignment, in the instruction set, that models the insertion of input values at run time by an external user.

4. Environment and state

In this section, we introduce different type of environments and states associated with SQL embedded application. Consider a database instance d consisting of three tables t_{emp} and t_{dept} and t_{prj}, as shown in Fig. 1.

4.1. Environment

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.

a

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

b

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

c

Prjno	Title
1	SQL
2	LatticeTheory
3	Semantics
4	Light

Fig. 1. Database d containing (a) t_{emp}, (b) t_{dept} and (c) t_{prj}.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

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, $\mathfrak{E}_a \triangleq \mathbb{V}_a \mapsto \mathfrak{D}_{\mathfrak{V}}$.

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

In the example depicted in Fig. 1, the index set I_x is {*emp*,*dept*,*prj*}, and the database d is the set $\{t_{emp}, t_{dept}, t_{prj}\}$. So, $\rho_d(emp) = t_{emp}$, for example.

Definition 3 (*Table environment*). Given a database environment ρ_d and a table $t \in d$. We define $attr(t) = \{a_1, a_2, \ldots, a_k\}$. So, $t \subseteq D_1 \times D_2 \times \cdots \times D_k$ 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 ith element of the l_jth row. In other words, ρ_t maps a_i to the ordered set of values over the rows of the table t.

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

Relation between database environment and table environment. Given a database d and a table $t_i \in d$ with $\vec{a} = attr(t_i)$. Then, $\rho_d(i) = \rho_{t_i}(\vec{a})$.

4.2. State and state transition

Given a SQL embedded program P, we define a state $\sigma \in \Im$ as a triplet $\langle I, \rho_d, \rho_a \rangle$ where $I \in \mathbb{I}$ 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,

$$\Im \triangleq \mathbb{I} \times \mathfrak{E}_d \times \mathfrak{E}_a$$

where $\mathfrak{E}_d$ denotes the set of all database environments, and $\mathfrak{E}_a$ denotes the set of all application environments. The set of states of a program P is defined as

$$\Im[\![P]\!] \triangleq P \times \mathfrak{E}[\![P]\!]$$

where $\mathfrak{E}[\![P]\!]$ is the set of environment of the program P whose domain is the set of program variables.

The state transition relation is defined as $\Gamma \triangleq \Im \mapsto \wp(\Im)$. The transitional semantics of a program P is, thus, defined as $\Gamma[\![P]\!] \triangleq \Im[\![P]\!] \mapsto \wp(\Im[\![P]\!])$.

In the next sections, we will describe in detail the semantic functions $E[\![\cdot]\!]$ and $B[\![\cdot]\!]$ for evaluating arithmetic and boolean expressions respectively, and $S[\![\cdot]\!]$ for evaluating SQL statements.

5. 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 $\exists t \in dom(\rho_d) : \; v_d = a_i \in attr(t)$ in
   $= E[\![v_d]\!](\rho_t, \rho_a)$
   $= \rho_t(a_i)$.
4. $E[\![v_d opc]\!](\rho_d, \rho_a)$ where, *op* represents the arithmetic operation.

   Let $\exists t \in dom(\rho_d) : \; v_d = a_i \in attr(t)$ and $op : \; D_i \times D_j \to D_k$ in
   $= E[\![v_d opc]\!](\rho_t, \rho_a)$
   $= \langle (mopc) \in D_k \mid m \in \rho_t(a_i) \wedge a_i \in D_i \wedge c \in D_j \rangle$.
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 $op : D_i \times D_j \to D_k$ in
   $= E[\![v_d op v_a]\!](\rho_t, \rho_a)$
   $= \langle (mopn) \in D_k \mid m \in \rho_t(a_i) \wedge \rho_a(v_a) = n \wedge a_i \in D_i \wedge v_a \in D_j \rangle$.
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
   $op : D_i \times D_j \to D_k$ in

$= E[\![v_{d_1} op v_{d_2}]\!](\rho_t,\rho_a)$
$= \langle m_r \in D_k | m_r = \pi_i(l_r) op \pi_j(l_r)$ where l_r is the rth row of $t \rangle$.

7. $E[\![e_1 \ op \ e_2]\!](\rho_d,\rho_a) =$

$$\begin{cases} \textbf{Case 1}: \ \exists!\ t \in dom(\rho_d): \text{ if } v_d \text{ occurs in } e_1 \text{ or } e_2 \text{ and } v_d = a \in attr(t) \\ = E[\![e_1 ope_2]\!](\rho_t,\rho_a) \\ = E[\![e_1]\!](\rho_t,\rho_a) op E[\![e_2]\!](\rho_t,\rho_a) \\ \textbf{Case 2}: \ \text{Let } T = \{t \in dom(\rho_d) | \exists v_d \text{ occurring in } e_1 \text{ or } e_2 \text{ s.t. } v_d = a \in attr(t)\} \\ \text{Let } T = \{t_1,t_2,\ldots,t_n\} \text{ and } t' = t_1 \times t_2 \times \cdots \times t_n \\ = E[\![e_1 ope_2]\!](\rho_{t'},\rho_a). \end{cases}$$

We generalize the arithmetic operation *op* on lists as follows: suppose *op* is a binary arithmetic operation over two lists S_1 and S_2. Also assume, $S' \subseteq S_1$, $s_1 \in S_1$ and $S'' \subseteq S_2$, $s_2 \in S_2$ with $|S'| = |S''|$, then the generalization of *op* is defined by,

$$op \triangleq \begin{cases} S' \ op \ s_2 = \langle s \ op \ s_2 | s \in S' \rangle \\ s_1 \ op \ S'' = \langle s_1 \ op \ s | s \in S'' \rangle \\ S' \ op \ S'' = \langle s_i' \ op \ s_i'' | s_i' \text{ and } s_i'' \text{ are the } i\text{th element of } S' \text{ and } S'' \text{ respectively} \rangle \end{cases}$$

Finally, the evaluation of boolean expressions is defined by

1. $B[\![true]\!](\rho_d,\rho_a) = true$.
2. $B[\![false]\!](\rho_d,\rho_a) = false$.
3. $B[\![e_1 \ op_r \ e_2]\!](\rho_d,\rho_a) = E[\![e_1]\!](\rho_d,\rho_a) \ op_r \ E[\![e_2]\!](\rho_d,\rho_a)$, where op_r represents the relational operator.
4. $B[\![\neg b]\!](\rho_d,\rho_a) = \neg B[\![b]\!](\rho_d,\rho_a)$.
5. $B[\![b_1 \vee b_2]\!](\rho_d,\rho_a) = B[\![b_1]\!](\rho_d,\rho_a) \vee B[\![b_2]\!](\rho_d,\rho_a)$.
6. $B[\![b_1 \wedge b_2]\!](\rho_d,\rho_a) = B[\![b_1]\!](\rho_d,\rho_a) \wedge B[\![b_2]\!](\rho_d,\rho_a)$.

6. Formal semantics of program instructions

Semantics $S[\![I]\!](\rho_d,\rho_a)$ of an instruction I in a SQL embedded program 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.

6.1. Semantics of SELECT statement

In this section, we describe the semantics of SELECT statement and we illustrate it with an example. We apologise for considering a quite complex SQL statement as example, but this way the reader should be able to get a more complete understanding of the transitions in the semantic functions.

Consider the database of Fig. 1 and the following SELECT statement C_{select}:

SELECT DISTINCT *Dno*, *Pno*, MAX(*Sal*), AVG(DISTINCT*Age*), COUNT(∗) FROM t_{emp} INTO v_a WHERE *Sal* > 1000 GROUP BY *Dno*,*Pno* HAVING MAX(*Sal*) < 4000 ORDER BY AVG(DISTINCT*Age*),*Dno*

An equivalent formulation of C_{select} is

SELECT DISTINCT⟨DISTINCT(Dno), DISTINCT(Pno), MAX○ALL(Sal), AVG○DISTINCT(*Age*), COUNT(∗)⟩) FROM t_{emp} INTO v_a WHERE *Sal* > 1000 GROUP BY ⟨*Dno*,*Pno*⟩ HAVING (MAX○ALL(*Sal*)) < 4000 ORDER BY ⟨AVG○DISTINCT(*Age*),*Dno*⟩

According to the abstract syntax, we get

- $\phi_1 = Sal > 1000$,
- $\vec{e} = \langle Dno, Pno \rangle$,
- $g(\vec{e}) =$ GROUP BY($\langle Dno, Pno \rangle$),
- $\phi_2 =$ (MAX○ALL(*Sal*)) < 4000,
- $\vec{h} =$ ⟨DISTINCT, DISTINCT,MAX○ALL,AVG○DISTINCT, COUNT⟩,
- $\vec{x} = \langle Dno, Pno, Sal, Age, * \rangle$,
- $\vec{h}(\vec{x}) =$ ⟨DISTINCT(*Dno*), DISTINCT(*Pno*),MAX○ALL(*Sal*), AVG○DISTINCT(*Age*), COUNT(∗)⟩,
- $\vec{e'} =$ ⟨AVG○DISTINCT(*Age*),*Dno*⟩, *where* AVG○DISTINCT(*Age*) simply represents an expression after the application of $\vec{h}(\vec{x})$,
- $f(\vec{e'}) =$ ORDERBYASC(⟨AVG○DISTINCT(*Age*), *Dno*⟩),
- $v_a =$ 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 DISTINCT(*Dno*),DISTINCT(*Pno*),MAX○ALL(*Sal*),AVG○DISTINCT(*Age*),COUNT(∗) respectively. For

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

instance, in java as a host language, v_a represents the object of the type *ResultSet*. Observe that, here we use the term INTOto understand the assignment into the application variable.

Thus, C_{select} is of the form as follows:
SELECT $r(\vec{h}(\vec{x}))$ FROM t_{emp} INTO $v_a(\vec{w})$ WHERE ϕ_1 GROUP BY $\vec{e}$ HAVING ϕ_2 ORDER BY ASC $\vec{e'}$
We now describe the semantics of SELECT statement step by step using the above example.
Recall from Table 1 that the syntax of SELECT statement is defined as

$$\langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})), \phi_1 \rangle$$

The semantics of SELECT statement is described below

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

Below the semantics of SELECT statement is unfolded step by step:
Step 1. Absorbing ϕ_1:

$$S[\![\langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})), \phi_1 \rangle]\!]_\varsigma(\rho_{t_0}, \rho_a) = S[\![\langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})), true \rangle]\!]_\varsigma(\rho_{t'}, \rho_a), \text{ where}$$

$$t' = \langle l_i \in t_0 | \text{let } var(\phi_1) = \vec{v'_d} \cup \vec{v'_a} \text{ with } \vec{v'_d} = \vec{a} \subseteq attr(t_0): \varsigma \vDash \phi_1[\, \pi_{\vec{a}}(l_i)/\vec{v'_d}][\rho_a(\vec{v'_a})/\vec{v'_a}] \rangle$$

Example. Since in our example $target(C_{select}) = \{t_{emp}\}$, we apply WHERE clause $\phi_1 = Sal > 1000$ on t_{emp}. The result is depicted in Table 2(a). The row "*eID*:7; *Name*:Alberto; *Age*:18; *Dno*:3; *Pno*:4; *Sal*:800; *Child-no*:1" is disregarded from the result because, $\varsigma \nvDash \phi_1[800/Sal]$. 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:

$$S[\![\langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), \phi, g(\vec{e})), true \rangle]\!]_\varsigma(\rho_t, \rho_a) = S[\![\langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), \phi, id), true \rangle]\!]_\varsigma(\rho_T, \rho_a)$$

where $g(\vec{e}) =$ Group By $(\vec{e})$ and $g(\vec{e})[t]$ is the maximal partition $T = \{t_1, t_2, \ldots, t_n\}$ of t s.t. $\forall t_i \in T$, $t_i \subseteq t$ and $\forall e_j \in \vec{e}$, $\forall m_k, m_l \in E[\![e_j]\!](\rho_{t_i}, \rho_a)$: $m_k = m_l$.

Example. Applying the grouping function $g(\vec{e}) =$ GROUP BY($\langle Dno, Pno \rangle$) on the result of step 1 based on the argument $\langle Dno, Pno \rangle$, we get four different partitions with $\langle 2,1 \rangle$, $\langle 1,2 \rangle$, $\langle 2,3 \rangle$ and $\langle 3,4 \rangle$ as the values of $\langle Dno, Pno \rangle$, depicted in Table 2(b).

Step 3. Absorbing ϕ:

$$S[\![\langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), \phi, id), true \rangle]\!]_\varsigma(\rho_T, \rho_a) = S[\![\langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), true, id), true \rangle]\!]_\varsigma(\rho_{T'}, \rho_a)$$

where T' is defined as follows: there is a sequence of functions $\vec{h'}$ occurring in ϕ, operating on groups, such that

$$\vec{h'}(\vec{x'}) \ni h_i'(x_i') \triangleq \begin{cases} \text{COUNT}(*) & \text{or} \\ \text{DISTINCT}(e) & \text{or} \\ s \circ r(e) \end{cases}$$

Let $\vec{v'_a}$ be a sequence of application variables occurring in ϕ and,

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

$$T' = \{t_i \in T | \varsigma \vDash \phi[\vec{c}_i / \vec{h'}(\vec{x'})][\rho_a(\vec{v'_a})/\vec{v'_a}]\}$$

Example. We apply the HAVING clause $\phi_2 =$ MAX ○ALL $(Sal) < 4000$ over all the groups in step 2. One group with the value of $\langle Dno, Pno \rangle$ equal to $\langle 2,3 \rangle$ 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 ○ALL(Sal) in ϕ_2 with the value which is returned by the function MAX ○ALL(Sal) applying on that group. The result is shown in Table 2(c).

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Table 2
Operations of C_{select}.

eID	*Name*	*Age*	*Dno*	*Pno*	*Sal*	*Child-no*
(a) Absorbing `WHERE` clause ϕ_1						
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
(b) `Grouping`						
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]						
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

Dno	*Pno*	*MAX*(*Sal*)	*AVG*(*DISTINCT Age*)	*COUNT*(∗)
(d) Performing $\vec{h}(\vec{x})$]				
2	1	2000	30	1
1	2	1900	34	3
3	4	3000	40	1
(e) Getting table out of the result from (d)				
2	1	2000	30	1
1	2	1900	34	3
3	4	3000	40	1
(f) Elimination of duplicates				
2	1	2000	30	1
1	2	1900	34	3
3	4	3000	40	1
(g) Ordering				
2	1	2000	30	1
1	2	1900	34	3
3	4	3000	40	1

(h) Assign to $\vec{v_a}$

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

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

$$= S[\![\langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), true, id), true \rangle]\!]_{\varsigma}(\rho_T, \rho_a) = S[\![\langle select(v_a, f(\vec{e'}), id, true, id), true \rangle]\!]_{\varsigma}(\rho_t, \rho_a) \quad \text{where}$$

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

As we mentioned earlier that the generation of maximal partitions T of the tuples depends on (i) whether the function g is present or not, (ii) $\vec{h}$ is $\vec{id}$ or not.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Example. In the example, we have $r(\vec{h}(\vec{x}))=$DISTINCT($\langle$ DISTINCT(Dno), DISTINCT(Pno), MAX ○ALL(Sal), AVG ○DISTINCT(Age), COUNT ($*$)$\rangle$). We perform $r(\vec{h}(\vec{x}))$ on each group resulting from step 3. We have three steps:

(a) Perform the ordered sequence of functions $\vec{h}(\vec{x})=\langle$ DISTINCT(Dno), DISTINCT(Pno), MAX ○ALL(Sal), AVG ○DISTINCT(Age), COUNT ($*$)$\rangle$ on each group: after applying $\vec{h}(\vec{x})$ on each group, we get the result as in Table 2(d).
(b) Get the table t out of these results obtained in step (a): this is shown in Table 2(e).
(c) Apply $r=$DISTINCTon the rows of table t obtained in step (b): we get Table 2(f) which is equal to the Table 2(e),since there is no duplicate rows.

Step 5. Possibly applying the ordering

$$S[\![\langle select(v_a, f(\vec{e}'), id, true, id), true\rangle]\!]_\varsigma(\rho_t,\rho_a)=S[\![\langle select(v_a, id, id, true, id), true\rangle]\!]_\varsigma(\rho_{t'},\rho_a), \quad \text{where } t'=f(\vec{e})[t]$$

Example. Performing $f(\vec{e}')=$ORDER BY ASC($\langle$ AVG ○DISTINCT(Age), $Dno\rangle$) on Table 2(f), we get Table 2(g).

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

$$S[\![\langle select(v_a, id, id, true, id), true\rangle]\!]_\varsigma(\rho_t,\rho_a)=(\rho_{t_0},\rho_{a'})$$

where $\rho_{a'}=\rho_a[v_a(\vec{w})/\rho_t(\vec{a})]$ with $\vec{a}=attr(t)$ and t_0 is the initial table of step 1. Here, the ith field $w_i\in\vec{w}$ *of* v_a is substituted by ith attribute $a_i\in\vec{a}$ of t.

Example. Finally, the result obtained in step 5 is assigned to the application variable v_a with fields $\vec{w}=\langle w_1, w_2, w_3, w_4, w_5\rangle$. The result is shown in Table 2(h).

6.2. Semantics of UPDATE statement

Consider the database of Fig. 1 and the following UPDATE statement C_{update}:

UPDATE t_{emp} SET $Age\coloneqq Age+2$, $Sal\coloneqq Sal+Sal\times 0.5$ WHERE $Sal>1500$.

According to the abstract syntax we get

- $\phi_1=Sal>1500$,
- $\vec{v_d}=\langle Age,Sal\rangle$,
- $\vec{e}=\langle Age+2,\ Sal+Sal\times 0.5\rangle$.

Thus, C_{update} is of the form as below:

UPDATE t_{emp} SET $\vec{v_d}\coloneqq\vec{e}$ WHERE ϕ.

Recall from Table 1 that the syntax of UPDATE statement is defined as

$$\langle update(\vec{v_d},\vec{e}),\phi\rangle$$

The semantics of UPDATE statement is described as follows: the update statement always targets an individual table. Let

$$target(\langle update(\vec{v_d},\vec{e}),\phi\rangle)=\{t\}$$

where $t\in dom(\rho_d)$. Therefore,

$$S[\![\langle update(\vec{v_d},\ \vec{e}),\ \phi\rangle]\!]_\varsigma(\rho_d,\rho_a)=S[\![\langle update(\vec{v_d},\ \vec{e}),\ \phi\rangle]\!]_\varsigma(\rho_t,\rho_a)$$

Below the semantics of UPDATE statement is unfolded step by step.
Step 1: Absorbing ϕ:

$$S[\![\langle update(\vec{v_d},\ \vec{e}),\ \phi\rangle]\!]_\varsigma(\rho_t,\rho_a)=S[\![\langle update(\vec{v_d},\ \vec{e}),\ true\rangle]\!]_\varsigma(\rho_{t'},\rho_a)$$

where

$$t'=\langle\ l_i\in t\,|\,\text{let } var(\phi)=\vec{v_d}\cup\vec{v_a}\quad\text{with } \vec{v_d}=\vec{a}\subseteq attr(t):\varsigma\vDash\phi[\pi_{\vec{a}}(l_i)/\vec{v_d}][\rho_a(\vec{v_a})/\vec{v_a}]\rangle$$

Example. In the example, $target(C_{update})=\{t_{emp}\}$. Applying WHERE clause $\phi=sal>1500$ on t_{emp}, we get the result t'_{emp} depicted in Table 3(a). Observe that two rows are disregarded as they do not satisfy the semantic structure ς of ϕ. That is, $\varsigma\nvDash\phi[1500/Sal]$ and $\varsigma\nvDash\phi[800/Sal]$.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Table 3
Operations of C_{update}.

eID	*Name*	*Age*	*Dno*	*Pno*	*Sal*	*Child*-no
(a) Table t'_{emp}: after absorbing WHERE clause ϕ						
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
(b) Table $t_{emp''}$: after update						
1	Matteo	32	2	1	3000	4
3	Joy	52	2	3	3450	3
4	luca	12	1	2	2550	1
5	Deba	42	3	4	4500	5
6	Andrea	72	1	2	2850	2
8	Bob	16	2	3	6000	3

Step 2: Update:

$$S[\![\langle update(\vec{v_d}, \vec{e}), true \rangle]\!]_{\varsigma}(\rho_t,\rho_a) = (\rho_{t'},\rho_a)$$

where

let $\vec{v_d} = \vec{a} \subseteq attr(t)$ and $\vec{e} = \langle e_1, e_2, \ldots, e_h \rangle$ and $E[\![\vec{e}]\!](\rho_t,\rho_a) = \langle \vec{m}_i \,|\, i = 1, \ldots, h \rangle$,
and let m_i^j be the jth element of the sequence $\vec{m}_i$ and a_i be the ith element of the sequence $\vec{a}$, and $t' = \langle l_j[m_i^j/a_i] \,|\, l_j \in t \rangle$.

Example. Performing the update operation $(\vec{v_d} := \vec{e}) = (\langle Age := Age+2, Sal := Sal+Sal \times 0.5 \rangle)$ on table t'_{emp} of step 1, we get the updated table t''_{emp} as shown in Table 3(b). Here, two expressions $(Age+2)$ and $(Sal+Sal \times 0.5)$ are evaluated over the environment $(\rho_{t'_{emp}}, \rho_a)$ first, and then for each rows of the table, two attributes '*Age*' and '*Sal*' are updated with the corresponding evaluated results respectively. Evaluation of the expression $(Age+2)$ over the environment $(\rho_{t'_{emp}}, \rho_a)$ gives the following results:

$$E[\![Age+2]\!](\rho_{t'_{emp}}, \rho_a) = \langle 32, 52, 12, 42, 72, 16 \rangle$$

$$E[\![Sal+Sal \times 0.5]\!](\rho_{t'_{emp}}, \rho_a) = \langle 3000, 3450, 2550, 4500, 2850, 6000 \rangle$$

Now the updation of the attribute '*Age*' is done for all rows as follows:

$$\langle l_1(32/Age), l_2(52/Age), l_3(12/Age), l_4(42/Age), l_5(72/Age), l_6(16/Age) \rangle$$

We do the same for $Sal := Sal+Sal \times 0.5$.

6.3. Semantics of INSERT statement

Consider the database of Fig. 1 and the following INSERT statement C_{insert}:

INSERT INTO t_{dept} VALUES (4,'*Electronins*','*Trieste*',2)

According to the abstract syntax we get

- $\vec{v_d} = \langle Deptno, Dname, Loc, MngrID \rangle$,
- $\vec{e} = \langle 4, $ '*Electronins*', '*Trieste*', $2 \rangle$.

Thus, C_{insert} is of the following form:

INSERT INTO $\vec{v_d}$ VALUES $\vec{e}$.

Recall from Table 1 that the syntax of INSERT statement is defined as

$$\langle insert(\vec{v_d}, \vec{e}), \phi \rangle$$

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Table 4
Operation of C_{insert}.

Deptno	*Dname*	*Loc*	*MngrID*
1	Math	Turin	4
2	Computer	Venice	1
3	Physics	Mestre	5
4	Electronins	Trieste	2

The semantics of INSERT statement is described as follows: the insert statement always targets an individual table. Let

$$t \in dom(\rho_d): \ target(\langle insert(\vec{v_d},\vec{e}),\phi\rangle) = \{t\}$$

Therefore,

$$S[\![\langle insert(\vec{v_d},\vec{e}),\phi\rangle]\!]_\varsigma(\rho_d,\rho_a) = S[\![\langle insert(\vec{v_d},\vec{e}),\phi\rangle]\!]_\varsigma(\rho_t,\rho_a) = S[\![\langle insert(\vec{v_d},\vec{e}),true\rangle]\!]_\varsigma(\rho_t,\rho_a) = (\rho_{t'},\rho_a)$$

where

let $\vec{v_d} = \vec{a} \subseteq attr(t)$, and $E[\![\vec{e}]\!](\rho_a) = \vec{x}$

$\vec{a} = \langle a_1,a_2,\ldots,a_n\rangle, \vec{x} = \langle x_1,x_2,\ldots,x_n\rangle$, and $l_{new} = \langle x_1/a_1,x_2/a_2,\ldots,x_n/a_n\rangle$, in

$t' = t \cup \{l_{new}\}$.

Observe that we suppose $\vec{v_d}$ includes all attributes of the table t. Although there exists alternative syntax where we can insert the values for selective attributes only, we can easily convert this alternative syntax into the one mentioned above by inserting undefined value $\mho$ in $\vec{e}$ for the unspecified attributes.

Example: In the example, $target(C_{insert}) = \{t_{dept}\}$. Since, $E[\![\vec{e}]\!](\rho_a) = \langle 4,$'*Electronics*','*Trieste*',$2\rangle$ and $\vec{v_d} = \vec{a} = \langle$*Deptno*, *Dname*,*Loc*,*MngrID*$\rangle$, we get $l_{new} = \langle 4/$*Deptno*,'*Electronics*'$/$*Dname*,'*Trieste*'$/$*Loc*,$2/$*MngrID*$\rangle$. After inserting the new row l_{new}, we get the resulting table $t_{dept}{}'$ as shown in Table 4 while the application environment ρ_a keeps unchanged.

6.4. Semantics of DELETE statement

Consider the database of Fig. 1 and the following DELETE statement C_{delete}:

DELETE FROM t_{emp} WHERE $Sal \geq 1800$

According to the abstract syntax we get

DELETE FROM t_{emp} WHERE ϕ

where ϕ represents the first-order formula "$Sal \geq 1800$".

Recall from Table 1 that the syntax of DELETE statement is defined as

$$\langle delete(\vec{v_d}),\phi\rangle$$

The semantics of DELETE statement is described as follows: the DELETE statement always targets an individual table. Let

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

Therefore,

$$S[\![\langle delete(\vec{v_d}),\phi\rangle]\!]_\varsigma(\rho_d,\rho_a) = S[\![\langle delete(\vec{v_d}),\phi\rangle]\!]_\varsigma(\rho_t,\rho_a) = (\rho_{t'},\rho_a)$$

where

$$t_d = \langle l_i \in t \mid \text{let } var(\phi) = \vec{v_d} \cup \vec{v_a} \text{ with } \vec{v_d} = \vec{a} \subseteq attr(t) : \varsigma \vDash \phi[\pi_a(l_i)/\vec{v_d}][\rho_a(\vec{v_a})/\vec{v_a}]\rangle$$

$$t' = t \backslash t_d$$

Observe that in case of DELETE, $\vec{v_d}$ includes all attributes of the table t.

Example: In the example, $target(C_{delete}) = \{t_{emp}\}$. Applying $\phi = Sal \geq 1800$ and deleting the rows which satisfy ϕ, we get t'_{emp} as shown in Table 5. Here, five rows are deleted from the table as they satisfy ϕ.

6.5. Formal semantics of non-SQL statements

$$S[\![skip]\!]_\varsigma(\rho_d,\rho_a) \triangleq (\rho_d,\rho_a)$$

$$S[\![v_a := e]\!]_\varsigma(\rho_d,\rho_a) \triangleq (\rho_d,\rho_a[E[\![e]\!](\rho_a)/v_a])$$

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Table 5
Operation of C_{delete}.

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~~

$S[\![v_a := ?]\!]_{\varsigma}(\rho_d,\rho_a) \triangleq ((\rho_d,\rho_{a'})$ where, b is any value in $dom(v_a)$ in $\rho_{a'} = \rho_a[b/v_a]$

7. Some inference rules for composite commands

The inference rules for composite instructions are obtained by induction

$$\frac{S[\![C_{sql_1}]\!](\rho_d,\rho_a) = t_1 \quad S[\![C_{sql_2}]\!](\rho_d,\rho_a) = t_2}{S[\![C_{sql_1} \texttt{ UNION } C_{sql_2}]\!](\rho_d,\rho_a) = t_1 \cup t_2}$$

$$\frac{S[\![C_{sql_1}]\!](\rho_d,\rho_a) = t_1 \quad S[\![C_{sql_2}]\!](\rho_d,\rho_a) = t_2}{S[\![C_{sql_1} \texttt{ INTERSECT } C_{sql_2}]\!](\rho_d,\rho_a) = t_1 \cap t_2}$$

$$\frac{S[\![C_{sql_1}]\!](\rho_d,\rho_a) = t_1 \quad S[\![C_{sql_2}]\!](\rho_d,\rho_a) = t_2}{S[\![C_{sql_1} \texttt{ MINUS } C_{sql_2}]\!](\rho_d,\rho_a) = t_1 \backslash t_2}$$

$$\frac{S[\![A_1]\!](\rho_d,\rho_a) = (\rho_{d'},\rho_{a'}) \quad S[\![A_2]\!](\rho_{d'},\rho_{a'}) = (\rho_{d''},\rho_{a''})}{S[\![A_1; A_2]\!](\rho_d,\rho_a) = (\rho_{d''},\rho_{a''})}$$

Consider the auxiliary conditional statement *cond*

$$cond(B[\![b]\!], S[\![A_1]\!], S[\![A_2]\!])(\rho_d,\rho_a) = (\rho_{d'},\rho_{a'})$$

where either $B[\![b]\!](\rho_d,\rho_a) = true$ and $S[\![A_1]\!](\rho_d,\rho_a) \triangleq (\rho_{d'},\rho_{a'})$ or $B[\![b]\!](\rho_d,\rho_a) = false$ and $S[\![A_2]\!](\rho_d,\rho_a) \triangleq (\rho_{d'},\rho_{a'})$.

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

$$S[\![\textit{if } b \textit{ then } A_1 \textit{ else } A_2]\!](\rho_d,\rho_a) = cond(B[\![b]\!], S[\![A_1]\!], S[\![A_2]\!])(\rho_d,\rho_a)$$

The semantics of the "*while b do A*" statement is expressed as follows: since "*while b do A*" ≡ "*if b then* (*A*; *while b do A*) *else skip*", we can write

$$S[\![\textit{while } b \textit{ do } A]\!](\rho_d,\rho_a) = S[\![\textit{if } b \textit{ then } (A; \textit{ while } b \textit{ do } A) \textit{ else skip}]\!](\rho_d,\rho_a) = FIX\ F$$

where $F(g) = cond(B[\![b]\!], g \circ S[\![A]\!], id)(\rho_d,\rho_a)$ and *FIX* is a fix-point operator.

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 if, $\{(\rho_x,\rho_{x'}) | S[\![I_1]\!](\rho_x) = \rho_{x'}\} = \{(\rho_x,\rho_{x'}) | S[\![I_2]\!](\rho_x) = \rho_{x'}\}$. In other words, $I_1 \equiv I_2$ if I_1 and I_2 determine the same partial function on states.

8. Soundness of the denotational semantics of SQL with respect to the standard semantics

The abstract syntax and the denotational semantics of SQL introduced in the previous sections correspond to the standard syntax and semantics of SQL as defined by ANSI [15] and the Relational Algebra. In particular, we can prove a correspondence between our denotational approach to the standard relational model approach to each SQL statement. For instance, consider the following basic SQL statement embedded in Java:

```
Statement stmt=conn.createStatement();
String Q="SELECT a1, a2, ..., an FROM t WHERE C";
ResultSet rs=stmt.executeQuery(Q);
```

where $a_1, a_2, \ldots, a_n$ represents the attributes of the table t and C is a condition.

An equivalent representation of the above SQL statement in Relational Algebra is

$t' = \sigma_C(t)$
$t'' = \pi_{\vec{a}}(t')$ where $\vec{a} = \langle a_1, a_2, \ldots, a_n \rangle$
$rs = t''$

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

In our proposed denotational approach, an equivalent formulation of the above java embedded SQL statement is shown below:

$$\langle select(v_a,f(\vec{e}),r(\vec{h}(\vec{x})),\phi_2,g(\vec{e})),\phi_1\rangle = \langle select(rs,id,\texttt{ALL}(\vec{id}(\vec{a})),true,id),C\rangle, \text{ where } \vec{id}(\vec{a}) = \langle id(a_1),id(a_2),\ldots,id(a_n)\rangle$$

Given an environment (ρ_d,ρ_a), the semantics is described as follows:

$$S[\![\langle select(rs,id,\texttt{ALL}(\vec{id}(\vec{a})),true,id),C\rangle]\!]_\varsigma(\rho_d,\rho_a) = S[\![\langle select(rs,id,\texttt{ALL}(\vec{id}(\vec{a})),true,id),C\rangle]\!]_\varsigma(\rho_t,\rho_a) \text{ where } t \in d$$
$$= S[\![\langle select(rs,id,\texttt{ALL}(\vec{id}(\vec{a})),true,id),true\rangle]\!]_\varsigma(\rho_{t'},\rho_a)$$

where

$$t' = \langle l_i \in t \mid \text{let } var(C) = \vec{v'_d} \cup \vec{v'_a} \text{ with } \vec{v'_d} = \vec{x} \subseteq attr(t) : \varsigma \vDash C[\pi_{\vec{x}}(l_i)/\vec{v'_d}][\rho_a(\vec{v'_a})/\vec{v'_a}]\rangle = \sigma_C(t)$$
$$= S[\![\langle select(rs,id,id,true,id),true\rangle]\!]_\varsigma(\rho_{t''},\rho_a) \quad (1)$$

where

$$t'' = \texttt{ALL}[\langle \vec{id}(E[\![\vec{a}]\!](\rho_{t'},\rho_a))\rangle] = E[\![\vec{a}]\!](\rho_{t'},\rho_a), \text{ since } \texttt{ALL does not remove or modify any element}$$
$$= \pi_{\vec{a}}(t'), \text{ according to the semantics of expressions} = (\rho_t,\rho_{a'}) \quad (2)$$

where

$$\rho_{a'}(rs) = t'' \quad (3)$$

Observe that Eqs. (1)–(3) show the correspondence between the Relational Algebra and Denotational semantic approaches.

9. Abstract semantics of SQL embedded applications

In this section, we lift the semantics of SQL operations defined so far to an abstract setting, where instead of working on the concrete databases, queries are applied to abstract databases, in which some information are disregarded and concrete values are possibly represented by suitable abstractions.

9.1. Abstract databases

Generally, traditional databases are *concrete databases* as they contain data from concrete domains, whereas *abstract databases* are obtained by replacing concrete values by the elements from abstract domains representing specific properties of interest. We may distinguish partial abstract database in contrast to fully abstract one, as in the former case only a subset of the data in the database is abstracted. The values of the data cells belonging to an attribute x are abstracted by following the Galois connection $(\wp(D_x^{con}),\alpha_x,\gamma_x,D_x^{abs})$ [7], where $\wp(D_x^{con})$ and D_x^{abs} represent the powerset of concrete domain of x and an abstract domain of x respectively, whereas α_x and γ_x represent the corresponding abstraction and concretization functions (denoted $\alpha_x : \wp(D_x^{con}) \to D_x^{abs}$ and $\gamma_x : D_x^{abs} \to \wp(D_x^{con})$) respectively. In particular, partial abstract databases are special case of fully abstract databases where abstraction and concretization functions for some attributes x are identity function *id*, and thus, follow the Galois connection $(\wp(D_x^{con}),id,id,\wp(D_x^{con}))$. Let us illustrate it by an example.

Example 1. The database in Fig. 1 consists of a concrete table t_{emp} that provides information about the employees of a company. We assume that the ages, salaries, and number of children of the employees lie between 5 and 100, between 500 and 10 000 and between 0 and 10 respectively. Considering an abstraction where ages and salaries of the employees are abstracted by the elements from the domain of intervals, and the number of children in the attribute '*Child-no*' are abstracted by the abstract values from the abstract domain $D^{abs}_{Child\text{-}no} = \{\bot, Zero, Few, Medium, Many, \top\}$ where $\top$ represents "*any*" and $\bot$ represents "*undefined*". The abstract table $t^{\sharp}_{emp}$ corresponding to t_{emp} *w.r.t.* these abstractions is shown in Table 6. Observe that the number of abstract tuples in an abstract database may be less than the number of tuples in the corresponding concrete database if the primary key is abstracted. The correspondence between concrete and abstract values of the attribute, for instance, '*Child-no*' can be formally expressed by the abstraction and concretization functions $\alpha_{child\text{-}no}$ and $\gamma_{child\text{-}no}$ respectively as follows:

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

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Table 6
Abstract table: $t^{\sharp}_{emp}$.

$eID^{\sharp}$	$Name^{\sharp}$	$Age^{\sharp}$	$Dno^{\sharp}$	$Pno^{\sharp}$	$Sal^{\sharp}$	$Child\text{-}no^{\sharp}$
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	[2500,10 000]	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	[2500,10 000]	Medium

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

We can similarly define the abstraction–concretization functions for other attributes as well. The corresponding abstract lattices for the attributes '*Age*', '*Sal*' and '*Child-no*' are shown in Fig. 2(a)–(c) respectively.

Definition 5 (*Abstract database*). Let dB be a database. The database $dB^{\sharp} = \alpha(dB)$ where α is an abstraction function, is said to be an abstract version of dB if there exists a representation function γ, called concretization function such that for all tuple $\langle x_1, x_2, \ldots, x_n \rangle \in dB$ there exists a tuple $\langle y_1, y_2, \ldots, y_n \rangle \in dB^{\sharp}$ such that $\forall i \in [1 \ldots n], x_i \in id(y_i) \vee x_i \in \gamma(y_i)$.

9.2. Syntax and semantics of statements in abstract domain

We now define the syntax and semantics of the SQL embedded applications in an abstract domain. We denote by the apex $^{\sharp}$, the syntactic elements of the abstract semantics. For each concrete element z, whenever we use the syntax $z^{\sharp}$, this means that there is a monotonic representation function γ from the abstract to the concrete domain such that $z \sqsubseteq \gamma(z^{\sharp})$.

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

$$C^{\sharp} ::= \langle A^{\sharp}, \phi^{\sharp} \rangle \mid C_1^{\sharp}\,\texttt{UNION}^{\sharp}\,C_2^{\sharp} \mid C_1^{\sharp}\,\texttt{INTERSECT}^{\sharp}\,C_2^{\sharp} \mid C_1^{\sharp}\,\texttt{MINUS}^{\sharp}\,C_2^{\sharp}$$

$$A^{\sharp} ::= select^{\sharp}(v_a^{\sharp}, f^{\sharp}(\vec{e^{\sharp}}), r^{\sharp}(h^{\sharp}(\vec{x^{\sharp}})), \phi^{\sharp}, g^{\sharp}(\vec{e^{\sharp}})) \mid update^{\sharp}(\vec{v_d^{\sharp}}, \vec{e^{\sharp}}) \mid insert^{\sharp}(\vec{v_d^{\sharp}}, \vec{e^{\sharp}}) \mid delete^{\sharp}(\vec{v_d^{\sharp}})$$

Arithmetic expressions over abstract domain are defined as expected, whereas boolean expressions are evaluated into a three-valued logics $\{true, false, \top\}$, where $\top$ means "*either true or false*".

$c^{\sharp} ::= n^{\sharp} \mid k^{\sharp}$.
$e^{\sharp} ::= c^{\sharp} \mid v_d^{\sharp} \mid v_a^{\sharp} \mid op^{\sharp} e^{\sharp} \mid e_1^{\sharp} op^{\sharp} e_2^{\sharp}$, where $op^{\sharp}$ represents abstract arithmetic operator.
$b^{\sharp} ::= e_1^{\sharp}\ op_r^{\sharp}\ e_2^{\sharp} \mid \neg b^{\sharp} \mid b_1^{\sharp} \vee b_2^{\sharp} \mid b_1^{\sharp} \wedge b_2^{\sharp} \mid true \mid false \mid \top$, where $op_r^{\sharp}$ represents abstract relational operator.

Abstract elements in abstract pre-condition $\phi^{\sharp}$ are defined as follows:

$\tau^{\sharp} ::= c^{\sharp} \mid v_a^{\sharp} \mid v_d^{\sharp} \mid f_n^{\sharp}(\tau_1^{\sharp}, \tau_2^{\sharp}, \ldots, \tau_n^{\sharp})$, where $f_n^{\sharp}$ is an abstract *n-ary* function.
$a_f^{\sharp} ::= R_n^{\sharp}(\tau_1^{\sharp}, \tau_2^{\sharp}, \ldots, \tau_n^{\sharp}) \mid \tau_1^{\sharp} = \tau_2^{\sharp}$, where $R_n^{\sharp}$ is an abstract *n-ary* relation: $R_n^{\sharp}(\tau_1^{\sharp}, \tau_2^{\sharp}, \ldots, \tau_n^{\sharp}) \in \{true, false, \top\}$.
$\phi^{\sharp} : = a_f^{\sharp} \mid \neg\phi_1^{\sharp} \mid \phi_1^{\sharp} \vee \phi_2^{\sharp} \mid \phi_1^{\sharp} \wedge \phi_2^{\sharp} \mid \forall x_i^{\sharp}\ \phi_1^{\sharp} \mid \exists x_i^{\sharp}\ \phi_1^{\sharp}$.

Different abstract functions involved in $A^{\sharp}$ are shown below:

$g^{\sharp} ::= \texttt{GROUP BY}^{\sharp} \mid id$
$r^{\sharp} ::= \texttt{DISTINCT}^{\sharp} \mid \texttt{ALL}^{\sharp}$
$s^{\sharp} ::= \texttt{AVG}^{\sharp} \mid \texttt{SUM}^{\sharp} \mid \texttt{MAX}^{\sharp} \mid \texttt{MIN}^{\sharp} \mid \texttt{COUNT}^{\sharp}$
$h^{\sharp}(e^{\sharp}) ::= s^{\sharp} \circ r^{\sharp}(e^{\sharp}) \mid \texttt{DISTINCT}^{\sharp}(e^{\sharp}) \mid id$
$h^{\sharp}(*) ::= \texttt{COUNT}^{\sharp}(*)$
$f^{\sharp} ::= \texttt{ORDER BY ASC}^{\sharp} \mid \texttt{ORDER BY DESC}^{\sharp} \mid id$

Instructions over an abstract domain are defined as follows:

$I^{\sharp} ::= skip \mid v_a^{\sharp} : = e^{\sharp} \mid v_a^{\sharp} : = ? \mid C^{\sharp} \mid if\ b^{\sharp}\ then\ I_1^{\sharp}\ else\ I_2^{\sharp} \mid while\ b^{\sharp}\ do\ I^{\sharp} \mid I_1^{\sharp}; I_2^{\sharp}$.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

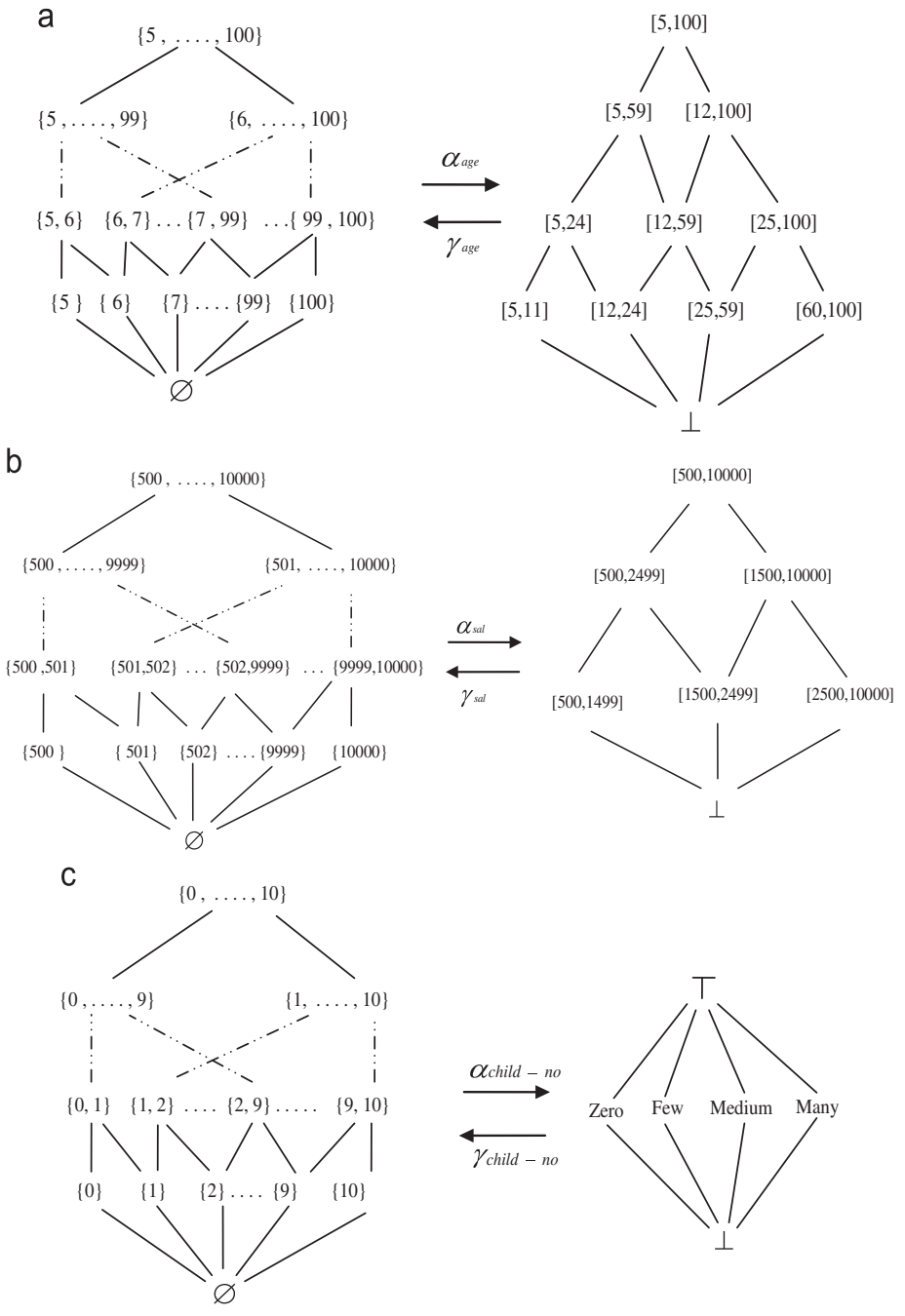

Fig. 2. Abstract lattices for attributes '*Age*', '*Sal*' and '*Child-no*'.

In the subsequent sections, we define abstract syntactic functions appearing in various abstract SQL statements so as to preserve the soundness in an abstract domain of interest. This way, we prove the soundness of abstract SQL statements with respect to their concrete counter-part. The soundness and completeness of an abstract function $f^\sharp$ are defined in Definition 6.

Definition 6. Let γ be a concretization function from an abstract domain to a concrete one. The soundness and completeness conditions for an abstract functions $f^\sharp$ with respect to the corresponding concrete function f are,

$f^\sharp$ is sound if $\gamma \circ f^\sharp \sqsupseteq f \circ \gamma$

$f^\sharp$ is complete if $\gamma \circ f^\sharp = f \circ \gamma$

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

9.3. Abstract pre-conditions

The pre-condition ϕ in C_{sql} follows first order logic, and are defined by the n-ary function f_n on constants and variables. Soundness (and completeness eventually) of its abstract version $f_n^\sharp$ relies on the local correctness of the operations in the abstract domain. For example, consider an abstract domain for parity represented by $PAR = \{\top, even, odd, \bot\}$. The '$\times$' operation over the concrete domain is mapped to its abstract version as follows: $odd(\times^\sharp)odd = odd$, $even(\times^\sharp)odd = even$, and $even(\times^\sharp)even = even$. Similarly, in case of abstract domain of sign represented by $SIGN = \{\top, +, -, \bot\}$, the corresponding operation would be $-(\times^\sharp)- = +$, $+(\times^\sharp)- = -$, and $+(\times^\sharp)+ = +$.

Given a set of terms $\{\tau_1, \ldots, \tau_n\}$, the relation $R_n(\tau_1, \ldots, \tau_n)$ appearing in ϕ results into either *true* or *false*. However, an abstract relation $R_n^\sharp(\tau_1^\sharp, \ldots, \tau_n^\sharp)$ corresponding to R_n follows three valued logic $\{true, false, \top\}$, where $\top$ represents either *true* or *false*. The correspondence between the relation R_n and its abstract version $R_n^\sharp$ should guarantee that, if $R_n^\sharp(\tau_1^\sharp, \ldots, \tau_n^\sharp)$ is true, then $\forall \tau_1 \in \gamma(\tau_1^\sharp), \ldots, \tau_n \in \gamma(\tau_n^\sharp)$: $R_n(\tau_1, \ldots, \tau_n)$ is true and if $R_n^\sharp(\tau_1^\sharp, \ldots, \tau_n^\sharp)$ is false, then $\forall \tau_1 \in \gamma(\tau_1^\sharp), \ldots, \tau_n \in \gamma(\tau_n^\sharp)$: $R_n(\tau_1, \ldots, \tau_n)$ is false. For instance, if we consider the binary relation '$<$' among integers, its abstract version '$<^\sharp$' on the domain of intervals is defined as follows:

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

Similarly, '$\ge^\sharp$' is defined as

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

Thus, abstract pre-condition $\phi^\sharp$ appearing in $C^\sharp$ identifies the set of active data from abstract database for which it evaluates to either *true* or $\top$.

Example 2. Consider the database of Fig. 1 containing a concrete table t_{emp} and consider the following SELECT statement:
$C_1 =$ `SELECT` *Age,Dno,Sal* `FROM` t_{emp} `WHERE` $Sal > 1600$.

If we execute C_1 on t_{emp}, we get the result ξ_1 shown in Table 7.

The abstract version of C_1, using the abstract mapping α defined in Example 1, is shown below:

$C_1^\sharp =$ `SELECT`$^\sharp$ $Age^\sharp, Dno^\sharp, Sal^\sharp$ `FROM` $t_{emp}^\sharp$ `WHERE` $Sal^\sharp >^\sharp [1500, 2499]$

where the abstract version $>^\sharp$ involved in the pre-condition over the domain of intervals is defined as follows:

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

Table 7
ξ_1: result of C_1 (concrete).

Age	*Dno*	*Sal*
30	2	2000
50	2	2300
10	1	1700
40	3	3000
70	1	1900
14	2	4000

Table 8
$\xi_1^\sharp$: result of $C_1^\sharp$.

$Age^\sharp$	$Dno^\sharp$	$Sal^\sharp$
[25,59]	2	[1500,2499]
[12,24]	1	[1500,2499]
[25,59]	2	[1500,2499]
[5,11]	1	[1500,2499]
[25,59]	3	[2500,10 000]
[60,100]	1	[1500,2499]
[12,24]	2	[2500,10 000]

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

The result of the abstract query $C_1^\sharp$ on the abstract table $t_{emp}^\sharp$ (Table 6) is depicted in Table 8. Observe that one row corresponding to $eID^\sharp = 7$ has been disregarded because the abstract well formed formula $[500, 1499] \geq^\sharp [1500, 2499]$ does not satisfy the semantic structure $\varsigma^\sharp$, as $1500 \geq 1499$ is true. Soundness is preserved, *i.e.* $\xi_1 \in \gamma(\xi_1^\sharp)$, as we include the rows (in this example, the row corresponding to $eID^\sharp = 2$) where the evaluation of the relation $\geq^\sharp$ yields $\top$; this might introduce inaccuracies, of course, in the abstract calculus, that results into a sound overapproximation of the concrete one.

9.4. Abstract syntactic functions in abstract SELECT statements

We now describe the correspondence between concrete and abstract functions involved in SELECT statement. Observe that many of these abstract functions differ from the corresponding concrete ones only on the domain and range, while their functionality are the same.

9.4.1. Abstract GROUP BY function

We denote by g the GROUP BY function in SELECT statement. The function $g(\vec{e})[t]$ where $\vec{e}$ represents an ordered sequence of arithmetic expressions, is applied on a table t and depending on the values of $\vec{e}$ over the tuples of the table t, it results into maximal partitions of the tuples in t. The tuples in the same partition will have the same values for $\vec{e}$, whereas the tuples in different partitions will have different values for $\vec{e}$. The GROUP BY function g is identity function id when no GROUP BY clause is present in SELECT statement. The function g and its abstract version $g^\sharp$ are shown below:

$$g ::= \texttt{GROUP BY} \mid id$$

$$g^\sharp ::= \texttt{GROUP BY}^\sharp \mid id$$

Abstract GROUP BY function $g^\sharp$ works in the similar way, but it is applied on abstract tables $t^\sharp$, instead of concrete ones. It partitions the abstract tuples of $t^\sharp$ based on the abstract values of $\vec{e^\sharp}$ over the tuples.

Lemma 1. *Let γ be a concretization function. The abstract* GROUP BY *function $g^\sharp$ is sound with respect to γ, i.e. $\gamma \circ g^\sharp \sqsupseteq g \circ \gamma$, where g is the concrete counter-part of $g^\sharp$.*

Proof. Let $t^\sharp$ be an abstract table and $\vec{e^\sharp}$ be an ordered sequence of abstract expressions. Let $t \in \gamma(t^\sharp)$ and $\vec{e} \in \gamma(\vec{e^\sharp})$, where γ is the concretization function.

Suppose $\{l_1, l_2, \ldots, l_n\}$ is a set of concrete partitions obtained from $g(\vec{e})[t]$, whereas $\{s_1, s_2, \ldots, s_m\}$ is the set of abstract partitions obtained from $g^\sharp(\vec{e^\sharp})[t^\sharp]$.

To prove the soundness of $g^\sharp$, we have to show that

$$\forall l_i, \exists s_j : \; l_i \subseteq \gamma(s_j) \text{ and } m \leq n$$

Consider a concrete partition $l_i \subseteq t$. From the Definition 5, we know that $\forall x \in t, \; \exists y \in t^\sharp : x \in \gamma(y)$. Thus, we have

$$\forall x_{i1}, x_{i2} \in l_i, \quad \exists y_{j1}^\sharp, y_{j2}^\sharp \in t^\sharp : \; x_{i1} \in \gamma(y_{j1}^\sharp) \wedge x_{i2} \in \gamma(y_{j2}^\sharp) \tag{4}$$

We know that the values of $\vec{e}$ for all tuples in a partition are same, *i.e.*

$$\forall x_{i1}, x_{i2} \in l_i, \quad \pi_{\vec{e}}(x_{i1}) = \pi_{\vec{e}}(x_{i2})$$

By the definition of abstraction, we get

$$\alpha(\pi_{\vec{e}}(x_{i1})) = \alpha(\pi_{\vec{e}}(x_{i2})) \text{ where } \alpha \text{ is abstraction function} \tag{5}$$

From Eqs. (4) and (5), we can write

$$\pi_{\vec{e^\sharp}}(y_{j1}^\sharp) = \pi_{\vec{e^\sharp}}(y_{j2}^\sharp) \tag{6}$$

Eq. (6) says that $y_{j1}^\sharp$ and $y_{j2}^\sharp$ belongs to the same partition $s_j \subseteq t^\sharp$, as the properties of $\vec{e^\sharp}$ in $y_{j1}^\sharp$ and $y_{j2}^\sharp$ are same. Therefore,

$$\forall x_{i1}, x_{i2} \in l_i \subseteq t, \quad \exists y_{j1}^\sharp, y_{j2}^\sharp \in s_j \subseteq t^\sharp : \; x_{i1} \in \gamma(y_{j1}^\sharp), x_{i2} \in \gamma(y_{j2}^\sharp)$$

or

$$\forall l_i, \; \exists s_j : \; l_i \subseteq \gamma(s_j)$$

Since an abstraction function α might be surjective, two different concrete partitions l_i and l_j $(i \neq j)$ might be mapped into the same abstract partition s_k, if

$$x_i \in l_i, \; x_j \in l_j, \; i \neq j : \; \alpha(\pi_{\vec{e}}(x_i)) = \alpha(\pi_{\vec{e}}(x_j))$$

Thus, the number of abstract partitions is less than or equal to the number of concrete partitions, *i.e.*, $m \leq n$. □

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

9.4.2. Abstract ALL and abstract DISTINCT

SELECT statement sometimes uses DISTINCT or ALL clause denoted by the function r which deals with duplicate tuples or duplicate values of expressions. Its abstract version $r^\sharp$ also works similarly, *i.e.*, deals with the duplicate elements in the list of abstract tuples or abstract values of expressions. The concrete function r and its abstract version are shown below:

$r ::= \texttt{DISTINCT} \mid \texttt{ALL}$
$r^\sharp ::= \texttt{DISTINCT}^\sharp \mid \texttt{ALL}^\sharp$

Lemma 2. *Let γ be a concretization function.* $\texttt{ALL}^\sharp$ *is complete, i.e.* $\gamma \circ \texttt{ALL}^\sharp = \texttt{ALL} \circ \gamma$.

Proof. When applying $\texttt{ALL}^\sharp$ to a list of abstract tuples $\langle l_i^\sharp : i \in I \rangle$, none of the tuple is removed or modified, and the same holds for ALL. Therefore,

$$\gamma \circ \texttt{ALL}^\sharp(\langle l_i^\sharp : i \in I\rangle) = \gamma(\texttt{ALL}^\sharp(\langle l_i^\sharp : i \in I\rangle)) = \gamma(\langle l_i^\sharp : i \in I\rangle) = \langle \gamma(l_i^\sharp) : i \in I\rangle$$
$$= \texttt{ALL}(\langle \gamma(l_i^\sharp) : i \in I\rangle) = \texttt{ALL}(\gamma(\langle l_i^\sharp : i \in I\rangle)) = \texttt{ALL} \circ \gamma(\langle l_i^\sharp : i \in I\rangle) \quad \square$$

Lemma 3. *If γ is injective, then* $\texttt{DISTINCT}^\sharp$ *is complete, i.e.* $\gamma \circ \texttt{DISTINCT}^\sharp = \texttt{DISTINCT} \circ \gamma$.

Proof. Suppose, after applying $\texttt{DISTINCT}^\sharp$ function on an abstract table $t^\sharp$, we get the abstract table $t_u^\sharp$ containing only unique rows. That means,

$$\forall l_1^\sharp, l_2^\sharp \in t_u^\sharp, \exists a^\sharp \in attr(t_u^\sharp) : \pi_{a^\sharp}(l_1^\sharp) \neq \pi_{a^\sharp}(l_2^\sharp)$$

In other words, any two rows in $t_u^\sharp$ differ by the property in at least one attribute position. Thus, concretization of $t_u^\sharp$ results into a concrete table t_u containing unique rows only, as γ is injective.

If we first apply γ on $t^\sharp$ before applying $\texttt{DISTINCT}^\sharp$, it results into a concrete table t containing duplicate rows if $t^\sharp$ has duplicate abstract rows. But after applying DISTINCT on t, we always get the same concrete table t_u. Therefore, the function $\texttt{DISTINCT}^\sharp$ is complete if γ is injective. $\square$

9.4.3. Abstract ORDER BY function

We denote by f the ORDER BY function appearing in SELECT statement. The operation $f(\vec{e})[t]$ sorts the tuples of the table t in ascending or descending order based on the values of $\vec{e}$ over those tuples. An abstract version $f^\sharp$ also works in similar way, but it is applied on abstract tables $t^\sharp$ and sorts the abstract tuples in ascending or descending order based on the abstract values of $\vec{e^\sharp}$ over the tuples in $t^\sharp$. The concrete functions f and their abstract versions are defined as

$f :: = \texttt{ORDER BY ASC} \mid \texttt{ORDER BY DESC} \mid id$
$f^\sharp :: = \texttt{ORDER BY ASC}^\sharp \mid \texttt{ORDER BY DESC}^\sharp \mid id$

Lemma 4. *If the representation function γ is monotone and injective, the functions $f^\sharp$ above are complete, i.e.* $\gamma \circ f^\sharp = f \circ \gamma$.

Proof. Given an abstract table $t^\sharp$ and an ordered sequence of abstract expressions $\vec{e^\sharp}$. Suppose for two tuples $l_i^\sharp, l_j^\sharp \in t^\sharp$, we have

$$\pi_{\vec{e^\sharp}}(l_i^\sharp) > \pi_{\vec{e^\sharp}}(l_j^\sharp) \tag{7}$$

Suppose $f^\sharp :: = \texttt{ORDER BY ASC}^\sharp$. Therefore, application of $f^\sharp$ sorts them in ascending order denoted by the ordered list $\langle l_j^\sharp, l_i^\sharp \rangle$. Since γ is injective, the concretization of this ordered list of abstract tuples always yield to an ordered list of concrete tuples denoted by $\langle l_j, l_i \rangle$, where $l_i \in \gamma(l_i^\sharp)$ and $l_j \in \gamma(l_j^\sharp)$.

Since concretization function γ is monotone, it preserves the ordering while mapping from abstract domain to concrete co-domain. Thus, from Eq. (7) we get

$$\gamma(\pi_{\vec{e^\sharp}}(l_i^\sharp)) > \gamma(\pi_{\vec{e^\sharp}}(l_j^\sharp))$$

or

$$\pi_{\vec{e}}(l_i) > \pi_{\vec{e}}(l_j) \tag{8}$$

where $\vec{e} \in \gamma(\vec{e^\sharp})$ and $l_i \in \gamma(l_i^\sharp)$ and $l_j \in \gamma(l_j^\sharp)$.

From Eq. (8), we get that the application of $f(::= \texttt{ORDER BY ASC})$ on l_i and l_j also yield to the same ordered list of concrete tuples *i.e.* $\langle l_j, l_i \rangle$.

Thus, $\gamma \circ f^\sharp$ will result into the same order of the elements as obtained by function $f \circ \gamma$. Hence, $f^\sharp$ is complete if γ is monotone and injective. $\square$

9.4.4. Abstract aggregate functions

In Section 3, we mentioned that the ordered sequence of functions $\vec{h}(\vec{e})$ where $\vec{h} \neq \vec{id}$, are applied on each partition obtained by GROUP BY function g, or on a single partition containing tuples for which pre-condition ϕ evaluates to *true* when no GROUP BY function is used. After performing $\vec{h}(\vec{e})$ on each partition, it results into a single concrete tuple.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

The aggregate functions MAX, MIN, AVG, SUM, COUNT appear in $h_i \in \vec{h}$, and are denoted by s. Aggregate functions return a single value when applied on a group of values.

Similarly, abstract aggregate functions $s^\sharp$ are applied on a set of abstract values, resulting into a single abstract value. The concrete aggregate functions s and its abstract version $s^\sharp$ are defined below:

$s ::= \texttt{AVG} \mid \texttt{SUM} \mid \texttt{MAX} \mid \texttt{MIN} \mid \texttt{COUNT}$
$s^\sharp ::= \texttt{AVG}^\sharp \mid \texttt{SUM}^\sharp \mid \texttt{MAX}^\sharp \mid \texttt{MIN}^\sharp \mid \texttt{COUNT}^\sharp$

Below we describe how to preserve the soundness of the SQL statements with aggregate functions over an abstract domain.

Given a set of concrete numerical values $X = \{a_1, a_2, \ldots, a_n\}$, the concrete aggregate functions s are defined as follows:

$$\texttt{AVG(X)} \triangleq \frac{\Sigma_1^n a_i}{n}$$

$$\texttt{SUM(X)} \triangleq \Sigma_1^n a_i$$

$$\texttt{MAX(X)} \triangleq a_i \in X, \text{ where } \forall j \in [1..n], i \neq j : \ a_i \geq a_j$$

$$\texttt{MIN(X)} \triangleq a_i \in X, \quad \text{where } \forall j \in [1..n], i \neq j : \ a_i \leq a_j$$

$$\texttt{COUNT(X)} \triangleq \#X, \quad \text{where \#denotes the cardinality of the set}$$

Corresponding to each s, we consider a concrete function fn equivalent to s, that is, $fn(X) \equiv s(X)$. The function fn and its abstract versions $fn^\sharp$ corresponding to the aggregate functions are defined as follows:

$fn ::= \texttt{average} \mid \texttt{summation} \mid \texttt{maximum} \mid \texttt{minimum} \mid \texttt{count}$
$fn^\sharp ::= \texttt{average}^\sharp \mid \texttt{summation}^\sharp \mid \texttt{maximum}^\sharp \mid \texttt{minimum}^\sharp \mid \texttt{count}^\sharp$

For instance, let $X^\sharp$ be a set of abstract values from the domain of intervals, *i.e.*, $X^\sharp = \{[l_i, h_i] \mid i \in [1 \cdots n], l_i, h_i \in \mathbb{Z}; l_i \leq h_i\}$. Let us denote $L = \{l_i \mid [l_i, h_i] \in X^\sharp\}$ and $H = \{h_i \mid [l_i, h_i] \in X^\sharp\}$. The abstract functions $fn^\sharp$ on $X^\sharp$ are defined as follows:

$$\texttt{average}^\sharp(X^\sharp) \triangleq [\texttt{average(L)}, \ \texttt{average(H)}]$$

$$\texttt{summation}^\sharp(X^\sharp) \triangleq [\texttt{summation(L)}, \ \texttt{summation(H)}]$$

$$\texttt{maximum}^\sharp(X^\sharp) \triangleq [\texttt{maximum(L)}, \ \texttt{maximum(H)}]$$

$$\texttt{minimum}^\sharp(X^\sharp) \triangleq [\texttt{minimum(L)}, \ \texttt{minimum(H)}]$$

$$\texttt{count}^\sharp(X^\sharp) \triangleq [\texttt{count(L)}, \ \texttt{count(H)}]$$

Formally, $fn^\sharp(X^\sharp) = [fn(L), \ fn(H)]$.

We already know that in abstract domain we select only those tuples for which the abstract pre-condition $\phi^\sharp$ evaluates to either *true* or $\top$. Thus, unlike concrete domain, the abstract groups on which abstract aggregate functions are applied contain a set of tuples that yield $\phi^\sharp$ to either *true* or $\top$. Let us denote by $G^\sharp$ an abstract group containing a set of abstract tuples. We can partition $G^\sharp$ into two parts: $G^\sharp_{yes}$ for which $\phi^\sharp$ evaluates to *true*, and $G^\sharp_{may}$ for which $\phi^\sharp$ evaluates to $\top$. Thus, we can write $G^\sharp = G^\sharp_{yes} \cup G^\sharp_{may}$. Observe that $G^\sharp_{yes} \cap G^\sharp_{may} = \emptyset$.

To ensure the soundness, the computation of abstract aggregate functions $s^\sharp$ on $G^\sharp$ are defined as follows: the result of $s^\sharp(e^\sharp)$ on $G^\sharp$ is denoted by an interval as below:

$$s^\sharp(e^\sharp)[G^\sharp] = [\texttt{min}^\sharp(a^\sharp), \ \texttt{max}^\sharp(b^\sharp)]$$

where

$$a^\sharp = fn^\sharp(e^\sharp)[G^\sharp_{yes}] \quad \text{and} \quad b^\sharp = fn^\sharp(e^\sharp)[G^\sharp]$$

By $fn^\sharp(e^\sharp)[G^\sharp_{yes}]$, we mean that function $fn^\sharp$ is applied on the set of abstract values obtained by evaluating $e^\sharp$ over the tuples in $G^\sharp_{yes}$, yielding a single abstract value as result. Similarly in $fn^\sharp(e^\sharp)[G^\sharp]$, $fn^\sharp$ is applied on the set of abstract values obtained by evaluating $e^\sharp$ over the tuples in $G^\sharp = G^\sharp_{yes} \cup G^\sharp_{may}$.

Both the functions $\texttt{min}^\sharp$ and $\texttt{max}^\sharp$ takes as parameter a single abstract value $a^\sharp$ and $b^\sharp$ respectively obtained from $fn^\sharp$, and returns a concrete numerical value as output. $\texttt{min}^\sharp(a^\sharp)$ returns the minimum concrete value from $\gamma(a^\sharp)$, whereas $\texttt{max}^\sharp(b^\sharp)$ returns the maximum concrete value from $\gamma(b^\sharp)$, where γ is the concretization function.

Example 3. Consider the database of Fig. 1 containing the concrete table t_{emp} and the following *SELECT* statement:

$C_2 = \texttt{SELECT AVG}(Age), Dno, \texttt{MAX}(Sal), \texttt{COUNT}(*) \ \texttt{FROM} \ t_{emp} \ \texttt{WHERE} \ Sal \geq 1500 \ \texttt{GROUP BY} \ Dno$

If we execute C_2 on t_{emp}, we get result ξ_2 shown in Table 9.

The abstract version of C_2 *i.e.* $C_2^\sharp$, using the abstract mapping α defined in Example 1, is defined as below:

$C_2^\sharp = \texttt{SELECT}^\sharp \ \texttt{AVG}^\sharp(Age^\sharp), Dno^\sharp, \texttt{MAX}^\sharp(Sal^\sharp), \texttt{COUNT}^\sharp(*) \ \texttt{FROM} \ t^\sharp_{emp} \ \texttt{WHERE} \ Sal^\sharp \geq^\sharp [1500, 2499] \ \texttt{GROUP BY}^\sharp \ Dno^\sharp$

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Table 9
ξ_2: result of C_2 (concrete).

AVG(*Age*)	*Dno*	*MAX*(*Sal*)	*COUNT*(∗)
34	1	1900	3
31.33	2	4000	3
40	3	3000	1

Table 10
$\xi_2^\sharp$: result of $C_2^\sharp$.

$AVG^\sharp(Age^\sharp)$	$Dno^\sharp$	$MAX^\sharp(Sal^\sharp)$	$COUNT^\sharp(*)$
[0, 45]	1	[0,2499]	[0, 3]
[12, 47.33]	2	[2500, 10 000]	[1, 3]
[25, 59]	3	[2500, 10 000]	[1, 1]

where '$\geq^\sharp$' involved in the pre-condition over the domain of intervals is defined as follows:

$$[l_i,h_i] \geq^\sharp [l_j,h_j] \triangleq \begin{cases} \textit{true if } l_i \geq h_j \\ \textit{false if } l_j > h_i \\ \top \quad \textit{otherwise} \end{cases}$$

The result of $C_2^\sharp$ on $t_{emp}^\sharp$ is shown in Table 10. Observe that, $G_{yes}^\sharp = \emptyset$ because $\phi^\sharp$ evaluates to $\top$ for all abstract tuples in the group with $Dno = 1$. Thus, $\texttt{AVG}^\sharp(Age^\sharp)$ is computed as follows:

$$a^\sharp = \texttt{average}^\sharp(\emptyset) = \text{NULL}$$

$$b^\sharp = \texttt{average}^\sharp([12, 24], [5, 11], [60, 100]) = [25.66, 45]$$

Since $\texttt{min}^\sharp(a^\sharp)$ and $\texttt{max}^\sharp(b^\sharp)$ return minimum value from $\gamma(a^\sharp)$ and maximum value from $\gamma(b^\sharp)$ respectively, we get $\texttt{min}^\sharp(a^\sharp) = \texttt{min}^\sharp(\text{NULL}) = 0$ and $\texttt{max}^\sharp(b^\sharp) = \texttt{max}^\sharp([25.66, 45]) = 45$. Thus, for group with $Dno = 1$, $\texttt{AVG}^\sharp(Age^\sharp) = [\texttt{min}^\sharp(a^\sharp), \texttt{max}^\sharp(b^\sharp)] = [0, 45]$.

Similarly, for the group with $Dno = 2$, first two tuples belong to $G_{may}^\sharp$, whereas last tuple belongs to $G_{yes}^\sharp$. Thus, $\texttt{MAX}^\sharp(Sal^\sharp)$ is computed as follows:

$$a^\sharp = \texttt{maximum}^\sharp([2500, 10\,000]) = [2500, 10\,000]$$

$$b^\sharp = \texttt{maximum}^\sharp([1500, 2499], [1500, 2499], [2500, 10\,000]) = [2500, 10\,000]$$

Thus, for group with $Dno = 2$, $\texttt{MAX}^\sharp(Sal^\sharp) = [\texttt{min}^\sharp(a^\sharp), \texttt{max}^\sharp(b^\sharp)] = [2500, 10\,000]$. Observe that abstraction is sound *i.e.* $\xi_2 \in \gamma(\xi_2^\sharp)$.

Lemma 5. *Let γ be a concretization function from the domain of intervals to a concrete numerical domain. The abstract functions $fn^\sharp$ are sound if they satisfy*

$$\gamma(fn^\sharp(X^\sharp)) \supseteq \{fn(X) \mid X \in \gamma(X^\sharp)\}$$

Proof. Let $X^\sharp$ be a set of abstract values from the domain of intervals, *i.e.*

$$X^\sharp = \{\, [l_i,h_i] \mid i \in [1..n];\ l_i,h_i \in \mathbb{Z};\ l_i \leq h_i \,\}$$

Consider two sets L and H, where $L = \{l_i \mid [l_i,h_i] \in X^\sharp\}$ and $H = \{h_i \mid [l_i,h_i] \in X^\sharp\}$. The abstract function $fn^\sharp$ *w.r.t.* the domain of intervals is defined on $X^\sharp$ as follows:

$$\texttt{average}^\sharp(X^\sharp) \triangleq [\texttt{average(L)}, \texttt{average(H)}]$$

$$\texttt{summation}^\sharp(X^\sharp) \triangleq [\texttt{summation(L)}, \texttt{summation(H)}]$$

$$\texttt{maximum}^\sharp(X^\sharp) \triangleq [\texttt{maximum(L)}, \texttt{maximum(H)}]$$

$$\texttt{minimum}^\sharp(X^\sharp) \triangleq [\texttt{minimum(L)}, \texttt{minimum(H)}]$$

$$\texttt{count}^\sharp(X^\sharp) \triangleq [\texttt{count(L)}, \texttt{count(H)}]$$

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Formally, we can write

$$fn^{\sharp}(X^{\sharp}) = [fn(L), fn(H)] \tag{9}$$

$$= [s(L), s(H)] \text{ since, } fn \equiv s \tag{10}$$

For a given set of concrete numerical values $X = \{a_1, a_2, \ldots, a_n\}$, the functions $fn \equiv s$ are defined as

$$\texttt{average(X)} \equiv \texttt{AVG(X)} \triangleq \frac{\Sigma_1^n a_i}{n}$$

$$\texttt{summation(X)} \equiv \texttt{SUM(X)} \triangleq \Sigma_1^n a_i$$

$$\texttt{maximum(X)} \equiv \texttt{MAX(X)} \triangleq a_i \in X, \text{ where } \forall j \in [1..n], i \neq j : \ a_i \geq a_j$$

$$\texttt{minimum(X)} \equiv \texttt{MIN(X)} \triangleq a_i \in X, \text{ where } \forall j \in [1..n], i \neq j : \ a_i \leq a_j$$

$$\texttt{count(X)} \equiv \texttt{COUNT(X)} \triangleq \#X, \text{ where \# denotes the cardinality of the set.}$$

Given two sets of numerical values $X = \{a_1, a_2, \ldots, a_n\}$ and $X' = \{a'_1, a'_2, \ldots, a'_n\}$. We say X is less than or equal to X' (denoted $X \sqsubseteq X'$) which is defined component-wise *i.e.* if $\forall i \in [1..n]$, $a_i \leq a'_i$, then $X \sqsubseteq X'$.

Since the function fn is monotone, we get

$$\text{if } X \sqsubseteq X' \quad \text{then } fn(X) \leq fn(X') \tag{11}$$

Let $X = \{b_i \mid [l_i, h_i] \in X^{\sharp}, l_i \leq b_i \leq h_i\} \in \gamma(X^{\sharp})$. Since, $\forall b_i \in X$ and $\forall [l_i, h_i] \in X^{\sharp}$: $l_i \leq b_i \leq h_i$, we can write

$$\forall X \in \gamma(X^{\sharp}) : \ L \sqsubseteq X \sqsubseteq H$$

According to Eq. (11),

$$\forall X \in \gamma(X^{\sharp}) : \ fn(L) \leq fn(X) \leq fn(H) \tag{12}$$

From Eqs. (9) and (12), we get

$$\forall X \in \gamma(X^{\sharp}) : \ fn(X) \in \gamma(fn^{\sharp}(X^{\sharp}))$$

or

$$fn(\gamma(X^{\sharp})) \subseteq \gamma(fn^{\sharp}(X^{\sharp}))$$

This implies that the abstract function $fn^{\sharp}$ is sound. □

Lemma 6. *Let γ be a concretization function from the domain of intervals to a concrete numerical domain. Abstract aggregate functions $s^{\sharp}$ are sound, i.e.*

$$\forall X \in X^{\sharp} : \ s(X) \in \gamma(s^{\sharp}(X^{\sharp}))$$

Proof. The computation of abstract aggregate functions $s^{\sharp}$ over a group of abstract tuples $G^{\sharp} = G_{yes} \cup G_{may}$ is defined as follows: $s^{\sharp}(e^{\sharp})[G^{\sharp}]$ is denoted by an interval

$$s^{\sharp}(e^{\sharp})[G^{\sharp}] = [\texttt{min}^{\sharp}(a^{\sharp}), \texttt{max}^{\sharp}(b^{\sharp})]$$

where

$$a^{\sharp} = fn^{\sharp}(e^{\sharp})[G^{\sharp}_{yes}] \quad \text{and} \quad b^{\sharp} = fn^{\sharp}(e^{\sharp})[G^{\sharp}]$$

From Lemma 5, we have that $fn^{\sharp}$ is sound, and therefore,

$$\forall G_y \in \gamma(G^{\sharp}_{yes}) : \ a \in \gamma(a^{\sharp}) \text{ or } a \geq \texttt{min}^{\sharp}(a^{\sharp})$$

where $a = fn(e)[G_y]$ and

$$\forall G \in \gamma(G^{\sharp}) : \ b \in \gamma(b^{\sharp}) \text{ or } b \leq \texttt{max}^{\sharp}(b^{\sharp})$$

where $b = fn(e)[G]$.

We know that $G^{\sharp} = G_{yes} \cup G_{may}$ contains the abstract tuples for which $\phi^{\sharp}$ evaluates to either *true* and $\top$. Given an abstract tuple $t^{\sharp} \in G^{\sharp}$ and abstract pre-condition $\phi^{\sharp}$, any concrete tuple $t \in \gamma(t^{\sharp})$ yield the corresponding concrete pre-condition $\phi \in \gamma(\phi^{\sharp})$ to either *true* or *false*, since we loose precision when moving from concrete to a domain of abstraction. Thus, $\forall G \in \gamma(G^{\sharp})$ where $G^{\sharp} = G_{yes} \cup G_{may}$, we can write

$$G = G_y \cup G_y{}' \cup G_f, \text{ where } G_y \in \gamma(G_{yes}) \text{ and } G_y{}' \cup G_f \in \gamma(G_{may})$$

where G_y and $G_y{}'$ are the set of concrete tuples for which ϕ evaluates to *true*, and G_f is the set of concrete tuples for which ϕ evaluates to *false*.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Since the concrete aggregate functions s are always applied on $(G_y \cup G_y')$ for which ϕ evaluates to *true*, we get

$$G_y \subseteq (G_y \cup G_y') \subseteq G$$

From the monotonicity property of *fn*, we get

$$fn(e)[G_y] \leq fn(e)[G_y \cup G_y'] \leq fn(e)[G]$$

or

$$fn(e)[G_y] \leq s(e)[G_y \cup G_y'] \leq fn(e)[G], \quad since\ fn \equiv s$$

or

$$a \leq s(X) \leq b$$

where X is obtained by evaluating e over the tuples of $(G_y \cup G_y')$.

or

$$\texttt{min}^\sharp(a^\sharp) \leq s(X) \leq \texttt{max}^\sharp(b^\sharp)$$

or

$$s(X) \in \gamma([\texttt{min}^\sharp(a^\sharp), \texttt{max}^\sharp(b^\sharp)])$$

or

$$s(X) \in \gamma(s^\sharp(X^\sharp))$$

Thus, the abstract aggregate functions $s^\sharp$ are sound. □

9.5. Abstract UPDATE, INSERT, and DELETE statements

The abstract semantics of UPDATE, INSERT and DELETE statements in an abstract domain of interest are defined below:

Abstract UPDATE statement: Let $C_{update} = \langle update(\vec{v_d},\vec{e}),\phi \rangle$ be an UPDATE statement with $target(C_{update}) = t$. Let $C^\sharp_{update} = \langle update^\sharp(\vec{v^\sharp_d},\vec{e^\sharp}),\phi^\sharp \rangle$ and $t^\sharp$ be their abstract versions in the abstract domain corresponding to C_{update} and t respectively, such that $target(C^\sharp_{update}) = t^\sharp$. According to the abstract semantics of $C^\sharp_{update}$, we get

$$S^\sharp[\![C^\sharp_{update}]\!](\rho_{t^\sharp},\rho_{a^\sharp}) = S^\sharp[\![\langle update^\sharp(\vec{v^\sharp_d},\vec{e^\sharp}),\phi^\sharp \rangle]\!](\rho_{t^\sharp},\rho_{a^\sharp}) = (\rho_{t^\sharp_1},\rho_{a^\sharp_1})$$

where

$$\rho_{t^\sharp_1}(x^\sharp) = \begin{cases} \rho_{t^\sharp \downarrow_t \phi^\sharp}(x^\sharp) \cup \rho_{t^\sharp \downarrow_u \phi^\sharp}(x^\sharp) \cup \rho_{t^\sharp \downarrow_f \phi^\sharp}(x^\sharp) & \text{if } x^\sharp \notin \vec{v^\sharp_d} \\ E^\sharp[\![e^\sharp_i]\!](\rho_{t^\sharp \downarrow_t \phi^\sharp},\rho_{a^\sharp}) \cup (\sqcup(E^\sharp[\![e^\sharp_i]\!](\rho_{t^\sharp \downarrow_u \phi^\sharp},\rho_{a^\sharp}),E^\sharp[\![x^\sharp]\!](\rho_{t^\sharp \downarrow_u \phi^\sharp}))) \cup \rho_{t^\sharp \downarrow_f \phi^\sharp}(x^\sharp) \\ \text{if } x^\sharp \text{ is the } i\text{th component of } \vec{v^\sharp_d} \quad \text{and } e^\sharp_i \text{ is the } i\text{th component of } \vec{e^\sharp} \end{cases}$$

By the notations $t^\sharp \downarrow_t \phi^\sharp$, $t^\sharp \downarrow_u \phi^\sharp$ and $t^\sharp \downarrow_f \phi^\sharp$ we denote the set of abstract tuples in $t^\sharp$ for which $\phi^\sharp$ evaluates to *true*, *unknown* and *false* respectively. The operator $\sqcup$ stands for computing least upper bound component-wise, *i.e.* $\sqcup(X^\sharp,Y^\sharp) = \{lub(x^\sharp_i,y^\sharp_i) \mid x^\sharp_i \in X^\sharp \wedge y^\sharp_i \in Y^\sharp\}$.

Abstract INSERT statement: Let $C^\sharp_{insert} = \langle insert^\sharp(\vec{v^\sharp_d},\vec{e^\sharp}),\phi^\sharp \rangle$ and $t^\sharp$ be an abstract INSERT statement and an abstract table corresponding to their concrete versions C_{insert} and t respectively, such that $target(C^\sharp_{insert}) = t^\sharp$. According to the abstract semantics of $C^\sharp_{insert}$, we get

$$S^\sharp[\![C^\sharp_{insert}]\!](\rho_{t^\sharp},\rho_{a^\sharp}) = S^\sharp[\![\langle insert^\sharp(\vec{v^\sharp_d},\vec{e^\sharp}),\phi^\sharp \rangle]\!](\rho_{t^\sharp},\rho_{a^\sharp}) = (\rho_{t^\sharp_1},\rho_{a^\sharp_1})$$

where

$$\text{let } \vec{v^\sharp_d} = \langle a^\sharp_1,a^\sharp_2,\ldots,a^\sharp_n \rangle = attr(t^\sharp), \text{ and } E^\sharp[\![\vec{e^\sharp}]\!](\rho_{a^\sharp}) = \vec{r^\sharp} = \langle r^\sharp_1,r^\sharp_2,\ldots,r^\sharp_n \rangle,$$

$$\text{and } l^\sharp_{new} = \langle r^\sharp_1/a^\sharp_1,r^\sharp_2/a^\sharp_2,\ldots,r^\sharp_n/a^\sharp_n \rangle, \text{ and } \rho_{t^\sharp_1}(\vec{x^\sharp}) = \rho_{t^\sharp \cup l^\sharp_{new}}(\vec{x^\sharp}).$$

Abstract DELETE statement: Given an abstract delete statement $C^\sharp_{delete} = \langle delete^\sharp(\vec{v^\sharp_d}),\phi^\sharp \rangle$ with $target(C^\sharp_{delete}) = t^\sharp$ corresponding to the concrete statement C_{delete} and concrete table t respectively. According to the abstract semantics of $C^\sharp_{delete}$, we get

$$S^\sharp[\![C^\sharp_{delete}]\!](\rho_{t^\sharp},\rho_{a^\sharp}) = S^\sharp[\![\langle delete^\sharp(\vec{v^\sharp_d}),\phi^\sharp \rangle]\!](\rho_{t^\sharp},\rho_{a^\sharp}) = (\rho_{t^\sharp_1},\rho_{a^\sharp_1})$$

where $\rho_{t^\sharp_1}(x^\sharp) = \rho_{t^\sharp \downarrow_u \phi^\sharp}(\vec{x^\sharp}) \cup \rho_{t^\sharp \downarrow_f \phi^\sharp}(\vec{x^\sharp})$.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

9.6. Soundness of abstract SQL statements

Given an abstraction, let T and $T^\sharp$ be a concrete and abstract table respectively. The correspondence between T and $T^\sharp$ are described using the concretization and abstraction maps γ and α respectively. If C_{sql} and $C^\sharp$ are representing the SQL queries on concrete and abstract domain respectively, let T_{res} and $T^\sharp_{res}$ are the results of applying C_{sql} and $C^\sharp$ on the T and $T^\sharp$ respectively. The following fact illustrate the soundness condition of abstraction:

$$\begin{array}{ccc} T & \xrightarrow{C_{sql}} & T_{res} \sqsubseteq \gamma(T^\sharp_{res}) \\ \uparrow{\scriptstyle \gamma} & & \uparrow{\scriptstyle \gamma} \\ T^\sharp & \xrightarrow[C^\sharp]{} & T^\sharp_{res} \end{array}$$

Lemma 7. *Let $T^\sharp$ be an abstract table and $C^\sharp$ be an abstract query. $C^\sharp$ is sound if* $\forall T \in \gamma(T^\sharp).\ \forall C_{sql} \in \gamma(C^\sharp):\ C_{sql}(T) \subseteq \gamma(C^\sharp(T^\sharp))$.

Proof. The computation of an abstract query $C^\sharp$ on an abstract table $T^\sharp$ can be defined as the computation of the composite function formed from its syntactic functional components. Consider the following abstract SELECT statement:

$$C^\sharp_{sel} = \langle\, select^\sharp(f^\sharp(\vec{e^\sharp}),\ r^\sharp(\vec{h^\sharp}(\vec{x^\sharp})),\ \phi^\sharp_1,\ g^\sharp(\vec{e^\sharp})), \phi^\sharp \,\rangle$$

and an abstract table $T^\sharp$ where $target(C^\sharp_{sel}) = T^\sharp$. We get the abstract result as follows:

$$\xi^\sharp = \mathrm{func}^\sharp_{sel}[T^\sharp] = (f^\sharp(\vec{e^\sharp}) \circ r^\sharp(\vec{h^\sharp}(\vec{x^\sharp})) \circ \phi^\sharp_1 \circ g^\sharp(\vec{e^\sharp}) \circ \phi^\sharp)[T^\sharp]$$

where $\mathrm{func}^\sharp_{sel} = f^\sharp(\vec{e^\sharp}) \circ r^\sharp(\vec{h^\sharp}(\vec{x^\sharp})) \circ \phi^\sharp_1 \circ g^\sharp(\vec{e^\sharp}) \circ \phi^\sharp$.

Let $C_{sel} \in \gamma(C^\sharp_{sel})$ and $T \in \gamma(T^\sharp)$. The computation of C_{sel} on T is defined as

$$\xi = \mathrm{func}_{sel}[T] = (\, f(\vec{e'}) \circ r(\vec{h}(\vec{x})) \circ \phi_1 \circ g(\vec{e})) \circ \phi)[T]$$

where $\mathrm{func}_{sel} = f(\vec{e'}) \circ r(\vec{h}(\vec{x})) \circ \phi_1 \circ g(\vec{e})) \circ \phi$.

We already proved that all syntactic abstract functional components in $C^\sharp_{sel}$ are sound with respect to their corresponding concrete counter-part. As the composition of sound abstract functions always yield to another sound abstract function, we get the abstract function $\mathrm{func}^\sharp_{sel}$ is sound *w.r.t.* func_{sel}. Thus, $C^\sharp_{sel}$ is sound, *i.e.* $\xi \in \gamma(\xi^\sharp)$. Similarly, we can prove the soundness for other SQL statements as well. Therefore, $\forall T \in \gamma(T^\sharp).\ \forall C_{sql} \in \gamma(C^\sharp):\ C_{sql}(T) \subseteq \gamma(C^\sharp(T^\sharp))$. □

9.7. Abstract UNION, INTERSECTION, MINUS operations

Given any abstract SQL statement $C^\sharp$, the result of it over an abstract database can be denoted by the tuple

$$\xi^\sharp = \langle\, \xi^\sharp_{yes},\ \xi^\sharp_{may} \,\rangle$$

where $\xi^\sharp_{yes}$ is the part of the result for which semantic structure of $\phi^\sharp$ evaluates to *true* and $\xi^\sharp_{may}$ represents the remaining part for which $\phi^\sharp$ evaluates to $\top$.[2]

Now we describe how to treat `UNION`, `INTERSECTION` and `MINUS` operation over an abstract domain so as to preserve the soundness.

9.7.1. Abstract UNION operation

Let, $C = C_l$ `UNION` C_r be a concrete query and dB be a concrete database. Let $\xi_l = [\![\, C_l \,]\!](dB)$ and $\xi_r = [\![\, C_r \,]\!](dB)$ be the result of the evaluation of C_l and C_r on dB. Clearly, $\xi = [\![\, C \,]\!](dB) = \xi_l \cup \xi_r$.

When we move from a concrete to an abstract domain of interest, let $C^\sharp_l$ and $C^\sharp_r$ be the corresponding abstract versions of C_l and C_r respectively. Let $dB^\sharp$ be an abstract database corresponding to dB *w.r.t.* this abstraction. We can denote the result of the execution of $C^\sharp_l$ and $C^\sharp_r$ on $dB^\sharp$ as follows:

$$\xi^\sharp_l = [\![\, C^\sharp_l \,]\!](dB^\sharp) = \langle\, \xi^\sharp_{yes_l}, \xi^\sharp_{may_l} \,\rangle$$

$$\xi^\sharp_r = [\![\, C^\sharp_r \,]\!](dB^\sharp) = \langle\, \xi^\sharp_{yes_r}, \xi^\sharp_{may_r} \,\rangle$$

The abstract version of C is defined as $C^\sharp = C^\sharp_l$ `UNION` $^\sharp$ $C^\sharp_r$, where the abstract union operation `UNION` $^\sharp$ is defined as:

$$\begin{aligned} \xi^\sharp &= [\![\, C^\sharp \,]\!](dB^\sharp) = [\![\, C^\sharp_l \text{ UNION}^\sharp\ C^\sharp_r \,]\!](dB^\sharp) = [\![\, C^\sharp_l \,]\!](dB^\sharp) \text{ UNION}^\sharp\ [\![\, C^\sharp_r \,]\!](dB^\sharp) = \xi^\sharp_l \text{ UNION}^\sharp\ \xi^\sharp_r \\ &= \langle\, \xi^\sharp_{yes_l}, \xi^\sharp_{may_l} \,\rangle \text{ UNION}^\sharp\ \langle\, \xi^\sharp_{yes_r}, \xi^\sharp_{may_r} \,\rangle = \langle\, (\xi^\sharp_{yes_l} \cup \xi^\sharp_{yes_r}), ((\xi^\sharp_{may_l} \cup \xi^\sharp_{may_r}) \setminus (\xi^\sharp_{yes_l} \cup \xi^\sharp_{yes_r})) \,\rangle \end{aligned}$$

[2] When SQL statement uses aggregate functions $s^\sharp$, application of $s^\sharp$ over a group $G^\sharp$ yields a single row in $\xi^\sharp$. This row belongs to $\xi^\sharp_{may}$ only if all rows of that group belong to $G^\sharp_{may}$, otherwise it belongs to $\xi^\sharp_{yes}$.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Observe that the first component of $\xi^{\sharp}$, *i.e.* $(\xi^{\sharp}_{yes_l} \cup \xi^{\sharp}_{yes_r})$ represents the *yes*-part of the result for which abstract pre-condition evaluates to *true*, whereas the second component $((\xi^{\sharp}_{may_l} \cup \xi^{\sharp}_{may_r}) \backslash (\xi^{\sharp}_{yes_l} \cup \xi^{\sharp}_{yes_r}))$ represents the *may*-part of the result for which the abstract pre-condition evaluates to $\top$.

Example 4. Consider the database of Fig. 1 that contains the concrete table t_{emp} and consider the following SELECT statement:

$C_3 = C_l$ UNION $C_r =$ SELECT * FROM t_{emp} WHERE *Age* > 15 UNION SELECT * FROM t_{emp} WHERE *Age* > 42
where $C_l =$ SELECT * FROM t_{emp} WHERE *Age* > 15,
and $C_r =$ SELECT * FROM t_{emp} WHERE *Age* > 42.

If we execute C_3 on t_{emp}, we get the result ξ_3 shown in Table 11.

By following the same abstraction and concretization mapping of Example 2, we get the abstract version of C_3 as follows:

$C^{\sharp}_3 = C^{\sharp}_l$ UNION $^{\sharp}$ $C^{\sharp}_r =$ SELECT $^{\sharp}$* FROM $t^{\sharp}_{emp}$ WHERE $Age^{\sharp} > {}^{\sharp}[12,24]$ UNION $^{\sharp}$ SELECT $^{\sharp}$* FROM $t^{\sharp}_{emp}$ WHERE $Age^{\sharp} > {}^{\sharp}[25,59]$

where '$>^{\sharp}$' appearing in pre-condition over the domain of intervals is defined as

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

The execution of the query $C^{\sharp}_l$ in Table 6 yields to the result shown in Table 12(a), where the tuples with $eID^{\sharp}$ equal to 2, 7, 8 belong to ξ_{may_l} and the tuples with $eID^{\sharp}$ equal to 1, 3, 5, 6 belong to ξ_{yes_l}. Similarly, the execution of the query $C^{\sharp}_r$ yields to the result shown in Table 12(b), where the tuples with $eID^{\sharp}$ equal to 1, 3, 5 belong to ξ_{may_r} and one tuple with $eID^{\sharp}$ equal to 6 belongs to ξ_{yes_r}. Thus, the result of abstract computation of $C^{\sharp}_3$ involving UNION $^{\sharp}$ is depicted in Table 12(c). Observe that

Table 11
ξ_3: result of C_3 (concrete).

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
5	Deba	40	3	4	3000	5
6	Andrea	70	1	2	1900	2
7	Alberto	18	3	4	800	1

Table 12
Abstract computation of $C^{\sharp}_3$.

$eID^{\sharp}$	$Name^{\sharp}$	$Age^{\sharp}$	$Dno^{\sharp}$	$Pno^{\sharp}$	$Sal^{\sharp}$	$Child\text{-}no^{\sharp}$
(a) $\xi^{\sharp}_l$: result of $C^{\sharp}_l$						
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
5	Deba	[25,59]	3	4	[2500,10 000]	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	[2500,10 000]	Medium
(b) $\xi^{\sharp}_r$: result of $C^{\sharp}_r$						
1	Matteo	[25,59]	2	1	[1500,2499]	Medium
3	Joy	[25,59]	2	3	[1500,2499]	Medium
5	Deba	[25,59]	3	4	[2500,10 000]	Many
6	Andrea	[60,100]	1	2	[1500,2499]	Few
(c) $\xi^{\sharp}_3$: resulting table after performing UNION $^{\sharp}$						
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
5	Deba	[25,59]	3	4	[2500,10 000]	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	[2500,10 000]	Medium

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

in the result $\xi_3^\sharp$, the *yes*-part $\xi^\sharp_{yes_3} = (\xi^\sharp_{yes_l} \cup \xi^\sharp_{yes_r})$ contains the tuples with $eID^\sharp$ equal to 1, 3, 5, 6 and the *may*-part $\xi^\sharp_{may_3} = ((\xi^\sharp_{may_l} \cup \xi^\sharp_{may_r}) \setminus (\xi^\sharp_{yes_l} \cup \xi^\sharp_{yes_r}))$ contains the tuples with $eID^\sharp$ equal to 2, 7, 8. Here the abstraction is sound *i.e.* $\xi_3 \in \gamma(\xi_3^\sharp)$.

9.7.2. Abstract INTERSECTION operation

Let, $\xi = [\![C]\!](dB)$ be the result of executing a concrete query C on a database dB, where $C = C_l$ INTERSECT C_r. It is clear that $\xi = \xi_l \cap \xi_r$, where $\xi_l = [\![C_l]\!](dB)$ and $\xi_r = [\![C_r]\!](dB)$, according to the concrete intersection operation INTERSECT.

Let $C_l^\sharp$, $C_r^\sharp$ and $dB^\sharp$ be abstract queries and abstract database corresponding to C_l, C_r and dB respectively *w.r.t.* an abstract domain of interest. Let $\xi_l^\sharp = [\![C_l^\sharp]\!](dB^\sharp) = \langle \xi^\sharp_{yes_l}, \xi^\sharp_{may_l} \rangle$ and $\xi_r^\sharp = [\![C_r^\sharp]\!](dB^\sharp) = \langle \xi^\sharp_{yes_r}, \xi^\sharp_{may_r} \rangle$.

The abstract version of C is, thus, defined as $C^\sharp = C_l^\sharp$ INTERSECT $^\sharp$ $C_r^\sharp$, where abstract intersection operation INTERSECT $^\sharp$ is defined as follows:

$$\begin{aligned}\xi^\sharp &= [\![C^\sharp]\!](dB^\sharp) = [\![C_l^\sharp \text{ INTERSECT}^\sharp\ C_r^\sharp]\!](dB^\sharp) = [\![C_l^\sharp]\!](dB^\sharp) \text{ INTERSECT}^\sharp\ [\![C_r^\sharp]\!](dB^\sharp) = \xi_l^\sharp \text{ INTERSECT}^\sharp\ \xi_r^\sharp \\ &= \langle \xi^\sharp_{yes_l}, \xi^\sharp_{may_l} \rangle \text{ INTERSECT}^\sharp \langle \xi^\sharp_{yes_r}, \xi^\sharp_{may_r} \rangle = \langle (\xi^\sharp_{yes_l} \cap \xi^\sharp_{yes_r}), ((\xi^\sharp_{may_l} \cap \xi_r^\sharp) \cup (\xi^\sharp_{may_r} \cap \xi_l^\sharp)) \rangle\end{aligned}$$

where the first component $(\xi^\sharp_{yes_l} \cap \xi^\sharp_{yes_r})$ represents the *yes*-part of the result, whereas the second component $((\xi^\sharp_{may_l} \cap \xi_r^\sharp) \cup (\xi^\sharp_{may_r} \cap \xi_l^\sharp))$ represents the *may*-part of the result.

Example 5. Consider the concrete table t_{emp} in Fig. 1 and the following SELECT statement:

$C_4 = C_l$ INTERSECTION C_r = SELECT * FROM t_{emp} WHERE $Age > 15$ INTERSECT SELECT * FROM t_{emp} WHERE $Age > 42$

where C_l = SELECT * FROM t_{emp} WHERE $Age > 15$,
and C_r = SELECT * FROM t_{emp} WHERE $Age > 42$.

If we execute C_4 on t_{emp}, we get the result ξ_4 shown in Table 13.

The corresponding abstract query $C_4^\sharp$, by following the same abstraction and concretization mapping of Example 2, is as follows:

$C_4^\sharp$ = SELECT $^\sharp$* FROM $t^\sharp_{emp}$ WHERE $Age^\sharp > {}^\sharp[12,24]$ INTERSECT $^\sharp$ SELECT $^\sharp$* FROM $t^\sharp_{emp}$ WHERE $Age^\sharp > {}^\sharp[25,59]$

The execution of the queries $C_l^\sharp$ and $C_r^\sharp$ on Table 6 yields to the result shown in Table 14(a) and (b) respectively. The result of abstract computation of $C_4^\sharp$ involving INTERSECT $^\sharp$ is depicted in Table 14(c), where the *yes*-part $\xi^\sharp_{yes_4} = (\xi^\sharp_{yes_l} \cap \xi^\sharp_{yes_r})$ contains only one tuple with $eID^\sharp$ equal to 6 and the *may*-part $\xi^\sharp_{may_4} = ((\xi^\sharp_{may_l} \cap \xi_r^\sharp) \cup (\xi^\sharp_{may_r} \cap \xi_l^\sharp))$ contains the tuples with

Table 13
ξ_4: result of C_4 (concrete).

eID	*Name*	*Age*	*Dno*	*Pno*	*Sal*	*Child-no*
3	Joy	50	2	3	2300	3
6	Andrea	70	1	2	1900	2

Table 14
Abstract computation of $C_4^\sharp$.

$eID^\sharp$	$Name^\sharp$	$Age^\sharp$	$Dno^\sharp$	$Pno^\sharp$	$Sal^\sharp$	$Child\text{-}no^\sharp$
(a) $\xi_l^\sharp$: result of $C_l^\sharp$						
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
5	Deba	[25,59]	3	4	[2500,10 000]	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	[2500,10 000]	Medium
(b) $\xi_r^\sharp$: result of $C_r^\sharp$						
1	Matteo	[25,59]	2	1	[1500,2499]	Medium
3	Joy	[25,59]	2	3	[1500,2499]	Medium
5	Deba	[25,59]	3	4	[2500,10 000]	Many
6	Andrea	[60,100]	1	2	[1500,2499]	Few
(c) $\xi_4^\sharp$: resulting table after performing INTERSECT $^\sharp$						
1	Matteo	[25,59]	2	1	[1500,2499]	Medium
3	Joy	[25,59]	2	3	[1500,2499]	Medium
5	Deba	[25,59]	3	4	[2500,10 000]	Many
6	Andrea	[60,100]	1	2	[1500,2499]	Few

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

$eID^\sharp$ equal to 1, 3, 5. Here the abstraction is sound *i.e.* $\xi_4 \in \gamma(\xi_4^\sharp)$.

9.7.3. Abstract MINUS operation

If we treat an abstract minus operation MINUS $^\sharp$ in a similar manner as of concrete MINUS, we cannot preserve the soundness. This happens due to the overapproximated results of the query on right side of MINUS $^\sharp$ operation that removes more information from the result of the query on the left side of MINUS $^\sharp$. So in order to preserve the soundness, we have to treat MINUS $^\sharp$ differently.

Consider an abstract SQL statement of the form $C^\sharp = C_l^\sharp$ *MINUS*$^\sharp$ $C_r^\sharp$. Let the result for $C_l^\sharp$ and $C_r^\sharp$ be $\xi_l^\sharp = \langle \xi^\sharp_{yes_l}, \xi^\sharp_{may_l} \rangle$ and $\xi_r^\sharp = \langle \xi^\sharp_{yes_r}, \xi^\sharp_{may_r} \rangle$ respectively.

The difference operation MINUS $^\sharp$ over an abstract domain is defined as follows:

$$\xi^\sharp = \xi_l^\sharp \text{ MINUS}^\sharp \ \xi_r^\sharp = \langle \xi^\sharp_{yes_l}, \xi^\sharp_{may_l} \rangle \text{ MINUS}^\sharp \ \langle \xi^\sharp_{yes_r}, \xi^\sharp_{may_r} \rangle = \langle (\xi^\sharp_{yes_l} \backslash (\xi^\sharp_{yes_l} \cap \xi^\sharp_{yes_r})), (\xi^\sharp_{may_l} \backslash (\xi^\sharp_{may_l} \cap \xi^\sharp_{yes_r})) \rangle$$

Observe that the first component $(\xi^\sharp_{yes_l} \backslash (\xi^\sharp_{yes_l} \cap \xi^\sharp_{yes_r}))$ represents the *yes*-part for which the abstract pre-condition strictly evaluates to *true*, whereas the second component $(\xi^\sharp_{may_l} \backslash (\xi^\sharp_{may_l} \cap \xi^\sharp_{yes_r}))$ represents the *may*-part for which the abstract pre-condition evaluates to $\top$.

Example 6. Consider the database of Fig. 1 that contains concrete table t_{emp} and consider the following SELECT statement:

$C_5 = C_l$ MINUS C_r = SELECT * FROM t_{emp} WHERE *Age* > 15 MINUS SELECT * FROM t_{emp} WHERE *Age* > 42
where C_l = SELECT * FROM t_{emp} WHERE *Age* > 15,
and C_r = SELECT * FROM t_{emp} WHERE *Age* > 42.

If we execute C_5 on t_{emp}, we get the result ξ_5 shown in Table 15.

By following the same abstraction and concretization mapping as of Example 2, we get the abstract version of C_5 as follows:

$C_5^\sharp = C_l^\sharp$ MINUS $^\sharp$ $C_r^\sharp$ = SELECT $^\sharp$* FROM $t^\sharp_{emp}$ WHERE $Age^\sharp > {}^\sharp[12,24]$ MINUS $^\sharp$ SELECT $^\sharp$*FROM $t^\sharp_{emp}$ WHERE $Age^\sharp > {}^\sharp[25,59]$

The execution of the query $C_l^\sharp$ and $C_r^\sharp$ in Table 6 yields to the results shown in Table 16(a) and (b) respectively. In Table 16(a), the tuples with $eID^\sharp$ equal to 2, 7, 8 belongs to $\xi^\sharp_{may_l}$, whereas the remaining four tuples belong to $\xi^\sharp_{yes_l}$. Similarly, in Table 16(b), the tuple with $eID^\sharp$ equal to 6 belongs to $\xi^\sharp_{yes_r}$, whereas the remaining three tuples belong to $\xi^\sharp_{may_r}$.

Table 15
ξ_5: result of C_5 (concrete).

eID	*Name*	*Age*	*Dno*	*Pno*	*Sal*	*Child-no*
1	Matteo	30	2	1	2000	4
2	Alice	22	1	2	1500	2
5	Deba	40	3	4	3000	5
7	Alberto	18	3	4	800	1

Table 16
Abstract computation of $C_5^\sharp$.

$eID^\sharp$	$Name^\sharp$	$Age^\sharp$	$Dno^\sharp$	$Pno^\sharp$	$Sal^\sharp$	$Child\text{-}no^\sharp$
(a) Result of $C_l^\sharp$						
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
5	Deba	[25,59]	3	4	[2500,10 000]	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	[2500,10 000]	Medium
(b) Result of $C_r^\sharp$						
1	Matteo	[25,59]	2	1	[1500,2499]	Medium
3	Joy	[25,59]	2	3	[1500,2499]	Medium
5	Deba	[25,59]	3	4	[2500,10 000]	Many
6	Andrea	[60,100]	1	2	[1500,2499]	Few
(c) $\xi_5^\sharp$: resulting table after performing MINUS $^\sharp$						
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
5	Deba	[25,59]	3	4	[2500,10 000]	Many
7	Alberto	[12,24]	3	4	[500,1499]	Few
8	Bob	[12,24]	2	3	[2500,10 000]	Medium

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Thus, $(\xi^{\sharp}_{yes_l} \backslash (\xi^{\sharp}_{yes_l} \cap \xi^{\sharp}_{yes_r}))$ contains the tuples with eID$^{\sharp}$ equal to 1, 3, 5, whereas $(\xi^{\sharp}_{may_l} \backslash (\xi^{\sharp}_{may_l} \cap \xi^{\sharp}_{yes_r})$ contains the tuples with eID$^{\sharp}$ equal to 2, 7, 8. The result of $C^{\sharp}_5$ involving MINUS $^{\sharp}$ is depicted in Table 16(c). Observe that the abstraction is sound *i.e.* $\xi_5 \in \gamma(\xi^{\sharp}_5)$.

9.8. Abstract control statements

Given an abstraction, the correspondence between the instructions I and its abstract versions $I^{\sharp}$ for the *conditional* and *while* statements are

- "*if b then* I_1 *else* I_2" is abstracted by

  if $(b^{\sharp} = true)$ *then* $I^{\sharp}_1$ *elseif* $(b^{\sharp} = false)$ *then* $I^{\sharp}_2$ *else* $I^{\sharp}_1 \sqcup I^{\sharp}_2$

- "*while b do I*" is abstracted by *FIX* $F^{\sharp}$ where $F^{\sharp}$ is the functional corresponding to the concrete *while* statement.

10. Formal semantics of SQL statements with co-related and non-co-related subquery

A subquery is a query that is nested inside a SELECT, UPDATE, INSERT, or DELETE statement, or inside another subquery. Subquery can be nested inside a WHERE or HAVING clause of an outer statement, or inside another subquery. A subquery can appear anywhere where an expression can be used, if it returns a single value. However, in practice, there is a limit on the levels of nesting based on the available memory and the complexity of the other expressions in the query.

Many queries can be evaluated by executing the subquery once and substituting the resulting value or values at the place of subquery.

In queries that include a co-related subquery (also known as a repeating subquery), the subquery depends on the outer query for its values. That means that the subquery is executed repeatedly, once for each row that might be selected by outer query.

If a table appears only in a subquery and not in the outer query, then the columns from that table cannot be included in the output.

The following example illustrates the co-related subquery which finds the name and location of those department under which the average salary of all employees is greater than or equal to 1000

SELECT *Dname,Loc* FROM t_{dept} WHERE $1000 \leq$ (SELECT AVG(*Sal*) FROM t_{emp} WHERE $t_{emp}.Dno = t_{dept}.Deptno$)

Here the subquery is *co-related* because the value of the subquery depends on the value of the attribute ($t_{dept}.Deptno$) which is the part of a table in the outer query.

But the following subquery is non-co-related:

SELECT *Dname, Loc* FROM t_{dept} WHERE *Deptno* = SOME (SELECT *Dno* FROM t_{emp} WHERE $Sal \geq 1500$)

Let C_{sql} be a query having C'_{sql} as a subquery. Suppose, $T^{out} = \{t_1, t_2, \ldots, t_n\}$ and $T^{in} = \{t'_1, t'_2, \ldots, t'_m\}$ are the set of tables explicitly appears in C_{sql} and C'_{sql} respectively, where $t^{out} = t_1 \times t_2 \times \cdots \times t_n$ and $t^{in} = t'_1 \times t'_2 \times \cdots \times t'_m$.

Definition 7 (*Co-related subquery*). C'_{sql} is co-related if $\exists x \in attr(t^{out})$ such that x used in C'_{sql}.

The syntax of the $C_{sql} = \langle A_{sql}, \phi \rangle$ with one level nested subquery is

1. $\langle select(v_a, f(\vec{e}'), r(\vec{h}(\vec{x})), \phi_2(C_{select}''), \vec{g}(\vec{e})), \phi_1(C'_{select}) \rangle$,
2. $\langle update(\vec{v_d}, \vec{e}), \phi(C_{select}) \rangle$,
3. $\langle insert(\vec{v_d}, \vec{e}), \phi(C_{select}) \rangle$,
4. $\langle delete, \phi(C_{select}) \rangle$,

where C_{select}, C'_{select} and C_{select}'' do not have any nested subquery.

We use the following idea to describe the semantics of SQL statement with co-related nested subquery.

Suppose t^{out} is partitioned into a set of mutual exclusive tables t^{out}_i, i ranges over the number of rows of t^{out}. Each table t^{out}_i contains a distinct row of t^{out}. So, if there are k rows in t^{out}, after partitioning we get k tables $t^{out}_1, t^{out}_2, \ldots, t^{out}_k$.

Now the following steps are executed k times for $i = 1, \ldots, k$:

1. $t_i = t^{out}_i \times t^{in}$.
2. Execute the subquery C'_{sql} on the environment (ρ_{t_i}, ρ_a) with $target(C'_{sql}) = t_i$.
3. Substitute the result obtained in step 2 at the place of the subquery C'_{sql} and execute the outer query C_{sql} on the environment $(\rho_{t^{out}_i}, \rho_a)$ with $target(C_{sql}) = t^{out}_i$.
4. Get the final result by taking union of all the results for all i obtained in step 3.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

10.1. SELECT statement with co-related subquery

10.1.1. Formal semantics of SELECT statement with co-related subqueries

$$S[\![\langle select(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), \phi_2(C''_{select}), \vec{g}(\vec{e})), \phi_1(C'_{select}) \rangle]\!]_\varsigma(\rho_d, \rho_a)$$
$$= S[\![\langle select(v_a, r(\vec{h}(\vec{x})), f(\vec{e'}), \phi_2(C''_{select}), \vec{g}(\vec{x}, \vec{e})), \phi_1(C'_{select}) \rangle]\!]_\varsigma(\rho_{t^{out}} \cup \rho_{t_1^{in}} \cup \rho_{t_2^{in}}, \rho_a)$$

where $target(C'_{select}) = \{t_1^{in}\}$ and $target(C_{select}{}'') = \{t_2^{in}\}$.

$$= \bigcup_i S[\![\langle select(v_a, r(\vec{h}(\vec{x})), f(\vec{e'}), \phi_2^i, \vec{g}(\vec{x}, \vec{e})), \phi_1^i \rangle]\!]_\varsigma(\rho_{t_i^{out}}, \rho_a), \text{ where}$$

Let $t_i^1 = t_i^{out} \times t_1^{in}$ and $t_i^2 = t_i^{out} \times t_2^{in}$, and $S[\![C'_{select}]\!](\rho_{t_i^1}, \rho_a) = t'_i$ and
$S[\![C''_{select}]\!](\rho_{t_i^2}, \rho_a) = t''_i$ with $target(C'_{select}) = \{t_i^1\}$, $target(C''_{select}) = \{t_i^2\}$, and
$\phi_1^i = \phi_1[\rho_{t_i}(a')/C'_{select}]$ and $\phi_2^i = \phi_2[\rho_{t'_i}(a'')/C''_{select}]$
with $a' = attr(t'_i)$ and $a'' = attr(t''_i)$

10.1.2. Illustration of the semantics of SQL statement with co-related subquery using an example

Consider the database instance in Fig. 3 and the following SELECT statement with a co-related subquery:

SELECT *Dname,Loc* FROM t_{dept} WHERE $1000 \leq$ (SELECTAVG(*Sal*) FROM t_{emp}WHERE $Dno = Deptno$)

From the above query we get the following information:

- $T^{out} = \{t_{dept}\}$, and thus, $t^{out} = t_{dept}$,
- $T^{in} = \{t_{emp}\}$, and thus, $t^{in} = t_{emp}$,
- Q= SELECT *Dname*, *Loc* FROM t_{dept} WHERE $1000 \leq (Q')$,
  where Q' = SELECT AVG(*Sal*) FROM t_{emp} WHERE $Dno = Deptno$.

Now we illustrate the operations step by step.

Step 1. Partition the table t^{out} into a set of tables t_i^{out} each containing one distinct row from t^{out}:
In the example, $t^{out} = t_{dept}$ with three rows. So, after partitioning we have three distinct tables t_1^{out}, t_2^{out}, t_3^{out} as shown in Table 17(a)–(c) respectively.

Step 2. Execute the following steps, for i=1, 2, 3:

Step (2a). Perform $t_i = t_i^{out} \times t^{in}$, i=1, 2, 3:
In the example, $t^{in} = t_{emp}$. Thus, for three partitions t_1^{out}, t_2^{out} and t_3^{out}, performing the above operation we get t_1, t_2 and t_3 respectively, as shown in Table 18(a)–(c).

a

eID	*Name*	*Age*	*Dno*	*Pno*	*Sal*
1	Matteo	28	2	1	2000
2	Stefano	30	1	2	1500
3	luca	25	1	2	1700
4	Alberto	35	3	4	800

b

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

Fig. 3. Database *dB*.

Table 17
Partitions of t_{dept}.

Deptno	*Dname*	*Loc*	*MngrID*
(a) t_1^{out}			
1	Math	Turin	4
(b) t_2^{out}			
2	Computer	Venice	1
(c) t_3^{out}			
3	Physics	Mestre	5

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Table 18
$t_i = t_i^{out} \times t_{emp}$ for $i = 1,2,3$.

Deptno	*Dname*	*Loc*	*MngrID*	*eID*	*Name*	*Age*	*Dno*	*Pno*	*Sal*
(a) $t_1 = t_1^{out} \times t_{emp}$									
1	Math	Turin	4	1	Matteo	28	2	1	2000
1	Math	Turin	4	2	Stefano	30	1	2	1500
1	Math	Turin	4	3	luca	25	1	2	1700
1	Math	Turin	4	4	Alberto	35	3	4	800
(b) $t_2 = t_2^{out} \times t_{emp}$									
2	Computer	Venice	1	1	Matteo	28	2	1	2000
2	Computer	Venice	1	2	Stefano	30	1	2	1500
2	Computer	Venice	1	3	luca	25	1	2	1700
2	Computer	Venice	1	4	Alberto	35	3	4	800
(c) $t_3 = t_3^{out} \times t_{emp}$									
3	Physics	Mestre	5	1	Matteo	28	2	1	2000
3	Physics	Mestre	5	2	Stefano	30	1	2	1500
3	Physics	Mestre	5	3	luca	25	1	2	1700
3	Physics	Mestre	5	4	Alberto	35	3	4	800

Table 19
Tables t_1', t_2' and t_3'.

	AVG(*Sal*)
(a) t_1'	1600
(b) t_2'	2000
(c) t_3'	800

Table 20
$S[[Q_i]](\rho_{t_i^{dept}}, \rho_a)$, for $i = 1,2,3$.

Dname	*Loc*
(a) Math	Turin
(b) Computer	Venice
(c) Empty	

Step (2b). Execute the inner query Q' on the environment (ρ_{t_i}, ρ_a) with $target(Q') = \{t_i\}$ and get the results as t_i', for $i = 1, 2, 3$:
In the example, we have three table environments ρ_{t_i} corresponding to t_i for $i = 1, 2, 3$. Thus, the execution of the inner query Q' over (ρ_{t_i}, ρ_a) yields to three results t_1',t_2' and t_3' respectively as follows:

(*i*) $t_1' = S[\![$ SELECT AVG(*Sal*) FROM t_1 WHERE *Dno* = *Deptno* $]\!](\rho_{t_1}, \rho_a)$,
(*ii*) $t_2' = S[\![$ SELECT AVG(*Sal*) FROM t_2 WHERE *Dno* = *Deptno* $]\!](\rho_{t_2}, \rho_a)$,
(*iii*) $t_3' = S[\![$ SELECT AVG(*Sal*) FROM t_3 WHERE *Dno* = *Deptno* $]\!](\rho_{t_3}, \rho_a)$.

The resulting tables t_1', t_2' and t_3' are shown in Table 19(a)–(c) respectively.

Step (2c). Substitute the result $\rho_{t_i'}(\vec{a})$ where $\vec{a} = attr(t_i')$, in place of subquery Q' and get the corresponding outer query Q_i with $target(Q_i) = \{t_i^{out}\}$ for $i = 1, 2, 3$:
In the example, $\rho_{t_i'}(\text{AVG}(Sal))$ returns three values 1600, 2000, and 800 for $i = 1, 2, 3$. So after substituting them in place of subquery Q' we get the following three final queries:

(*i*) $Q_1 =$ SELECT *Dname*, *Loc* FROM t_1^{out} WHERE $1000 \leq (1600)$,
(*ii*) $Q_2 =$ SELECT *Dname*, *Loc* FROM t_2^{out} WHERE $1000 \leq (2000)$,
(*iii*) $Q_3 =$ SELECT *Dname*, *Loc* FROM t_3^{out} WHERE $1000 \leq (800)$.

Step (2d). Execute Q_i over the environment $(\rho_{t_i^{out}}, \rho_a)$, for $i = 1, 2, 3$:
The execution of Q_1, Q_2 and Q_3 over $(\rho_{t_1^{dept}}, \rho_a)$, $(\rho_{t_2^{dept}}, \rho_a)$ and $(\rho_{t_3^{dept}}, \rho_a)$ gives the results shown in Table 20(a)–(c) respectively. Observe that the execution in the third case results into an empty table.

Step (2e). Get the final result of the query $Q = \bigcup_i S[\![Q_i]\!](\rho_{t_i^{out}}, \rho_a)$, for $i = 1, 2, 3$:
In the example, $S[\![Q]\!](\rho_{t^{out}} \cup \rho_{t^{in}}, \rho_a) = S[\![Q]\!](\rho_{t_{dept}} \cup \rho_{t_{emp}}, \rho_a) = S[\![Q_1]\!](\rho_{t_1^{out}}, \rho_a) \cup S[\![Q_2]\!](\rho_{t_2^{out}}, \rho_a) \cup S[\![Q_3]\!](\rho_{t_3^{out}}, \rho_a)$.
The result is shown in Table 21

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Table 21
$S[[Q]](\rho_{t^{out}} \cup \rho_{t_{in}}, \rho_a)$.

Dname	*Loc*
Math	Turin
Computer	Venice

10.2. SELECT statement with non-co-related subquery

10.2.1. Formal semantics of SELECT statement with non-co-related subqueries

$$S[\![\langle select(v_a, f(\vec{e}'), r(\vec{h}(\vec{x})), \phi_2(C_{select_2}), \vec{g}(\vec{e})), \phi_1(C_{select_1}) \rangle]\!]_\varsigma(\rho_d, \rho_a)$$

$$= S[\![\langle select(v_a, f(\vec{e}'), r(\vec{h}(\vec{x})), \phi_2(C_{select_2}), \vec{g}(\vec{e})), \phi_1(C_{select_1}) \rangle]\!]_\varsigma(\rho_{t^{out}} \cup \rho_{t_1^{in}} \cup \rho_{t_2^{in}}, \rho_a)$$

where $target(C_{select_1}) = \{t_1^{in}\}$ and $target(C_{select_2}) = \{t_2^{in}\}$. $= S[\![\langle select(v_a, f(\vec{e}'), r(\vec{h}(\vec{x})), \phi_2', \vec{g}(\vec{e})), \phi_1' \rangle]\!]_\varsigma(\rho_{t^{out}}, \rho_a)$, where

$S[\![C_{select_1}]\!](\rho_{t_1^{in}}, \rho_a) = t'$ and $S[\![C_{select_2}]\!](\rho_{t_2^{in}}, \rho_a) = t''$, and $\phi_1' = \phi_1[\rho_{t'}(a')/C_{select_1}]$ and $\phi_2' = \phi_2[\rho_{t''}(a'')/C_{select_2}]$

with $a' = attr(t')$ and $a'' = attr(t'')$

10.2.2. Illustration of the semantics of SELECT statement with non-co-related subquery using an example

Consider the following SELECT statement with non-co-related subquery and the database instance *dB* as depicted in Fig. 3:

SELECT *Dname*, *Loc* FROM t_{dept} WHERE *Deptno* = SOME(SELECT *Dno* FROM t_{emp} WHERE *Sal* ≥ 1500)

From the above query we get the following informations:

- $T^{out} = \{t_{dept}\}$, and thus, $t^{out} = t_{dept}$,
- $T^{in} = \{t_{emp}\}$, and thus, $t^{in} = t_{emp}$,
- $Q =$ SELECT *Dname*, *Loc* FROM t_{dept} WHERE *Deptno* = *SOME*(Q'), where $Q' =$ SELECT *Dno* FROM t_{emp} WHERE *Sal* ≥ 1500.

Now we illustrate the operations step by step.

Step 1. Execute the inner query Q' on the environment $(\rho_{t^{in}}, \rho_a)$ and get the result as t':
In the example $t^{in} = t_{emp}$ and thus, the semantics execution of Q' on $(\rho_{t_{emp}}, \rho_a)$ yields to the resulting table t', shown in Table 22.

Step 2. Substitute the result $\rho_{t'}(\vec{a})$ with $\vec{a} = att(t')$ in the place of subquery Q', and get the corresponding outer query Q:
As $\rho_{t'}(Dno)$ return the $\langle 2, 1, 1 \rangle$. Substituting it at the place of Q', we get

$Q =$ SELECT *Dname*, *Loc* FROM t_{dept} WHERE *Deptno* = SOME(2, 1, 1)

Step 3. Execute Q over the environment $(\rho_{t^{out}}, \rho_a)$:
The execution of Q over $(\rho_{t^{out}}, \rho_a)$, *i.e.* $S[\![Q]\!](\rho_{t_{dept}}, \rho_a)$ yields to the result shown in Table 23.

10.3. Formal semantics of UPDATE/INSERT/DELETE statement with co-related subquery

$$S[\![\langle A_{sql}, \phi(C_{select}) \rangle]\!]_\varsigma(\rho_d, \rho_a) = S[\![\langle A_{sql}, \phi(C_{select}) \rangle]\!]_\varsigma(\rho_{t^{out}} \cup \rho_{t^{in}}, \rho_a) \text{ where } target(C_{select}) = \{t^{in}\}$$

$$= \bigcup_i S[\![\langle A_{sql}, \phi_i \rangle]\!]_\varsigma(\rho_{t_i^{out}}, \rho_a)$$

where

Let $t_i = t_i^{out} \times t^{in}$, and $S[\![C_{select}]\!](\rho_{t_i}, \rho_a) = t_i$ with $target(C_{select}) = \{t_i\}$, and $\phi_i = \phi[\rho_{t_i}(a)/C_{select}]$ with $a = attr(t_i)$.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

Table 22
Table $t' = S[\![Q']\!](\rho_{t_{emp}}, \rho_a)$.

Dno
2
1
1

Table 23
$S[\![Q]\!](\rho_{t_{dept}}, \rho_a)$.

Dname	*Loc*
Math	Turin
Computer	Venice

10.4. Formal semantics of UPDATE/INSERT/DELETE statement with non-co-related subquery

$$S[\![\langle A_{sql}, \phi(C_{select}) \rangle]\!](\rho_d, \rho_a) = S[\![\langle A_{sql}, \phi(C_{select}) \rangle]\!](\rho_{t^{out}} \cup \rho_{t^{in}}, \rho_a)$$
$$= S[\![\langle A_{sql}, \phi' \rangle]\!](\rho_{t^{out}}, \rho_a) \text{ where } S[\![C_{select}]\!](\rho_{t^{in}}, \rho_a) = t \text{ and } \phi' = \phi[\rho_t(a)/C_{select}] \text{ with } a = attr(t)$$

11. Applications

The abstraction of relational database system has many interesting application areas. Let us discuss in detail a few of them.

11.1. Observation-based fine grained access control

The granularity of traditional access control mechanism is coarse-grained and can be applied at database or table level only. The Fine Grained Access Control (FGAC) mechanisms [3], on the other hand, provides the safety of the database information even at lower level such as individual tuple level or cell level. However, in traditional FGAC, the notion of sensitivity of the information is too restrictive (either public or private) and impractical in some real systems where intensional leakage of the information to some extent is allowed with the assumption that the power of the external observer is bounded. Thus, we need to weaken or downgrading the sensitivity level of the database information, hence, consider a weaker attacker model. The weaker attacker model characterizes the observational characteristics of the attacker and can be able to observe specific properties of the private data. For instance, suppose the database in an online transaction system contains credit card numbers for its customers. According to the disclosure policy, the employees of the customer-care section are able to see the last four digits of the credit card numbers, whereas all the other digits are completely hidden. The traditional FGAC policy is unable to implement this type of security framework without changing the database structure. To cope with this situation, we introduced an Observation-Based Fine Grained Access Control (OFGAC) mechanism [16] for Relational Database Management System (RDBMS) based on the Abstract Interpretation framework. In this setting, data are made accessible at various level of abstraction based on their sensitivity. For instance, the credit card number "3456 1985 5672 1856", according to the policy, will be viewed as "**** **** **** 1856". Therefore, unauthorized users are not able to infer the exact content of a cell containing confidential information, while they are allowed to get partial information out of it, according to their access rights. Recently, we extended this OFGAC framework to the context of XML documents [17].

11.2. Persistent watermarking

In the existing watermarking schemes of relational databases [18], the watermark is generated and embedded based on the database content. As a result, the watermark verification phases completely rely on the database content. In other words, the success of the watermark detection is content dependent. Benign updates or any other intensional processing of this content may damage or distort the existing watermark, leading the detection phase almost infeasible. In [19,20], we address the issue of persistency of watermarks that serves as a way to recognize the integrity and ownership proof of the database, while allowing the evaluation of the database content by queries in a set of queries Q. The proposed algorithms generate the watermark by exploiting the information in the static part of the database states and the invariants of the database information represented by semantics-based properties. The static part contains the data cells of the database state that are not affected by the queries in Q at all. The semantics-based properties are extracted from a form of abstract

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

database, and these include Intra-cell (IC), Intra-tuple (IT), and Intra-attribute among-tuples (IA) properties. The IC property deals with the properties of individual cells obtained from non-relational abstract domain. The IT property deals with the inter-relation between two or more attribute values in the same tuple represented by the elements from relational abstract domain (e.g., the domain of octagons, polyhedra, etc.). The IA property is obtained from the set of independent tuples and can be represented by either relational or non-relational abstract domain.

11.3. Cooperative query answering

The traditional query processing system requires the users to have precise information about the problem domain, database schema, and database content. It always provides limited and exact answers, or even no information at all when the exact information is not available in the database. To remedy such shortcomings, the notion of cooperative query answering [1,2] has been explored as an effective mechanism that provides users an intelligent database interface to issue approximate queries independent to the underlying database structure and its content, and supplies additional useful information as well as the exact answers. As an example, in response to the query about "specific flight departing at *10 a.m.* from *Rome Fiumicino* airport to *Paris Orly* airport", the cooperative query answering system may return "all flight information during *morning* from airports in *Rome* to airports in *Paris*", and thus, the user will be able to choose other flights from nearby airports if specific flight is unavailable. Searching approximate values for a specialized value is equivalent to find an abstract value of the specialized value, since the specialized values of the same abstract value constitute approximate values of one another. In [21], we introduced three key issues: soundness, relevancy and optimality of the cooperative answers, which can be used as a milestone to compare different cooperative schemes in the literature. In addition, we proposed a cooperative scheme based on the Abstract Interpretation framework to address these key issues as well.

11.4. Static analysis framework for the transactions to optimize the integrity constraint checking

In [22,23], the authors introduce the way to optimize integrity constraints checking for a transaction at compile-time to reduce the run-time overhead. They consider only the object-oriented databases where the initial databases are represented in the form of first order logic formulas (treated as abstract form of databases). They use predicate transformer as a way to provide the abstract interpretation of the transactions so as to collect the run-time behavior of the transaction at compile-time. Our proposal, similarly, can serve as a semantics-based static analysis framework for the applications or transactions that interact with relational databases.

11.5. Property-based querying and approximate query answering

Other applications include property-based querying where users are mostly interested in answers based on some specific properties of the database information, rather than their exact content. Also abstraction of database system can serve as a way to answering queries approximately in order to reduce query response times, when the precise answer is not necessary or early feedback is helpful [24].

12. Related works

Most popular commercial and open source databases currently in use are based on the relational data model which serves as a formal basis for relational database system. The relational model of data was first proposed by Codd in 1970 [25]. In [26], Codd defines a collection of operations on relations which is defined as Relational Algebra.

Relational Calculus is based on a branch of mathematical logic called predicate calculus. In 1967, possibly, Kuhns first used the idea of predicate calculus as the basis for query language in [27]. The applied form of predicate calculus specifically tailored to relational databases was proposed by Codd [26]. A language explicitly based on that calculus called "*Data Sublanguage ALPHA*" was also presented by Codd in [28].

In [11], the authors describe the formal semantics of SQL using a formal model, called Extended Three Valued Predicate Calculus (E3PVC). This model is basically based on a set of rules that determine a syntax-driven translation of SQL queries. These rules allow the transformation of a general E3VPC expression to a canonical form, which can be manipulated using traditional, two-valued predicate calculus and solves the equivalence of SQL queries.

The Relational Algebra and Relational Calculus can be used as a model for designing many approaches to query optimization. Many works [8,12,29,10] on semantics and optimization of SQL queries focussed on the approach of translating SQL to a formal language.

Bultzingwloewen [8] gives a precise definition of the semantics of SQL queries having aggregate functions and identifies some problems associated with optimization along with their solutions. Here the semantics is defined by translating SQL queries into extended relational calculus expressions based on the work in [9] where the extension of relational algebra and relational calculus is achieved by including aggregate functions only in a natural manner. Moreover, he considers the NULL values as well in the extended version of relational algebra and relational calculus. He proved that these extended

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

relational calculus and relational algebra are equivalent and have the same expressive power. He also discussed a new general processing strategy for aggregate functions.

In [12], the authors proposed a syntax directed translator of SQL into relational algebra. This is done in two steps: transform SQL queries into equivalent SQL queries accepted by restricted grammar and then transform the restricted one into relational algebra expression. The translator can be in conjunction with an optimizer which operates on the relational algebra expression. This translator defines the semantics of the SQL language and can be used for the proof of equivalence of SQL queries with different syntactic forms. Translation of SQL into equivalent relational algebra via relational calculus is also presented in [29]. However, this translation is not optimized.

13. 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, (iii) static analysis of the transactions to optimize the integrity constraint checking, (iv) to provide observation-based fine grained access control to the database information, (v) approximate query answering, (vi) persistent watermarking, (vii) to provide users either partial view or "customized replicas" of the database, etc.

Acknowledgment

Work partially supported by RAS L.R. 7/2007 Project TESLA.

References

[1] Chu WW, Chen Q. A structured approach for cooperative query answering. IEEE Transactions on Knowledge and Data Engineering 1994;6:738–49.
[2] Keun Shin M, Huh S-Y, Lee W. Providing ranked cooperative query answers using the metricized knowledge abstraction hierarchy. Expert Systems with Applications 2007;32:469–84.
[3] Wang Q, Yu T, Li N, Lobo J, Bertino E, Irwin K, Byun J-W. On the correctness criteria of fine-grained access control in relational databases. In: Proceedings of the 33rd international conference on very large data bases (VLDB'07). Vienna, Austria: VLDB Endowment; 2007. p. 555–66.
[4] 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 4th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'77). Los Angeles, CA, USA: ACM Press; 1977. p. 238–52.
[5] Cousot P, Cousot R. Systematic design of program analysis frameworks. In: Proceedings of the 6th annual ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL'79). San Antonio, Texas: ACM Press; 1979. p. 269–82.
[6] Cousot P, Cousot R. Systematic design of program transformation frameworks by abstract interpretation. In: Proceedings of the 29th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'02). Portland, OR, USA: ACM Press; 2002. p. 178–90.
[7] Giacobazzi R, Ranzato F, Scozzari F. Making abstract interpretations complete. Journal of the ACM 2000;47:361–416.
[8] Bultzingwloewen GV. Translating and optimizing SQL queries having aggregates. In: Proceedings of the 13th international conference on very large data bases (VLDB'87). Brighton, England: Morgan Kaufmann Publishers Inc.; 1987. p. 235–43.
[9] Klug A. Equivalence of relational algebra and relational calculus query languages having aggregate functions. Journal of the ACM 1982;29:699–717.
[10] Nakano R. Translation with optimization from relational calculus to relational algebra having aggregate functions. ACM Transactions on Database Systems 1990;15:518–57.
[11] Negri M, Pelagatti G, Sbattella L. Formal semantics of SQL queries. ACM Transactions on Database System 1991;17:513–34.
[12] Ceri S, Gottlob G. Translating SQL into relational algebra: optimization, semantics, and equivalence of SQL queries. IEEE Transactions on Software Engineering 1985;11:324–45.
[13] Halder R, Cortesi A. Abstract interpretation for sound approximation of database query languages. In: Proceedings of the IEEE 7th international conference on informatics and systems (INFOS'10). Cairo, Egypt: IEEE Press; 2010. p. 53–9 [IEEE Catalog Number: IEEE CFP1006J-CDR].
[14] Goldrei D. Propositional and predicate calculus: a model of argument. Springer; 2005.
[15] A.N.S. Institute. Information technology-database languages-SQL-part 2: foundation (SQL/foundation).
[16] Halder R, Cortesi A. Observation-based fine grained access control for relational databases. In: Proceedings of the 5th international conference on software and data technologies (ICSOFT'10). Athens, Greece: INSTICC Press; 2010. p. 254–65.
[17] Halder R, Cortesi A. Observation-based fine grained access control for xml documents. In: Proceedings of the 10th international conference on computer information systems and industrial management applications (CISIM'11), vol. 245. Kolkata, India: Springer CCIS; 2011. p. 267–76.
[18] Halder R, Pal S, Cortesi A. Watermarking techniques for relational databases: survey, classification and comparison. Journal of Universal Computer Science 2010;16:3164–90.
[19] Halder R, Cortesi A. A persistent public watermarking of relational databases. In: Jha S, Mathuria A, editors. Proceedings of the 6th international conference on information systems security (ICISS'10), Lecture Notes in Computer Science, vol. 6503. Gandhinagar, Gujarat, India: Springer; 2010. p. 216–30.
[20] Halder R, Cortesi A. Persistent watermarking of relational databases. In: Proceedings of the 1st IEEE international conference on advances in communication, network, and computing (CNC'10). Calicut, Kerala, India: IEEE Computer Society; 2010. p. 46–52.
[21] Halder R, Cortesi A. Cooperative query answering by abstract interpretation. In: Proceedings of the 37th international conference on current trends in theory and practice of computer science (SOFSEM'11), vol. 6543. Nový Smokovec, Slovakia: Springer LNCS; 2011. p. 284–96.
[22] Benzaken V, Schaefer X. Ensuring efficiently the integrity of persistent object systems via abstract interpretation. In: Proceedings of the 7th workshop on persistent object systems, Cape May, New Jersey, USA, p. 72–87.
[23] Benzaken V, Schaefer X. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In: Proceedings of the 11th European conference on object-oriented programming (ECOOP'97). Finland: Springer-Verlag LNCS; 1997. p. 60–84.
[24] Ioannidis YE, Poosala V. Histogram-based approximation of set-valued query answers. In: Proceedings of the 25th international conference on very large data bases. Edinburgh, Scotland, UK: Morgan Kaufmann Publishers Inc.; 1999. p. 174–85.
[25] Codd EF. A relational model of data for large shared data banks. Communications of the ACM 1983;25th Anniversary Issue 26:64–9.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004

[26] Codd EF. Relational completeness of database sublanguages. Database Systems 1972:65–98.
[27] Kuhns JL. Answering questions by computer: a logical study. In: Report RM-5428-PR. Santa Monica, California: The Rand Corporation; 1967. p. 137.
[28] Codd EF. A database sublanguage founded on the relational calculus. In: Proceedings of 1971 ACM-SIGFIDET workshop on data description, access and control. San Diego, California: ACM Press; 1971. p. 35–68.
[29] Jarke M, Koch J. Query optimization in database systems. ACM Computing Surveys (CSUR) 1984;16:111–52.

Please cite this article as: Halder R, Cortesi A. Abstract interpretation of database query languages. Computer Languages, Systems & Structures (2011), doi:10.1016/j.cl.2011.10.004