TPLP **19** (5–6): 808–825, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000206

Applying Constraint Logic Programming to SQL Semantic Analysis[*]

FERNANDO SÁENZ-PÉREZ
Complutense University of Madrid, 28040 Madrid, Spain
(*e-mail:* fernan@sip.ucm.es)

submitted 25 July 2019; accepted 31 July 2019

Abstract

This paper proposes the use of Constraint Logic Programming (CLP) to model SQL queries in a data-independent abstract layer by focusing on some semantic properties for signalling possible errors in such queries. First, we define a translation from SQL to Datalog, and from Datalog to CLP, so that solving this CLP program will give information about inconsistency, tautology, and possible simplifications. We use different constraint domains which are mapped to SQL types, and propose them to cooperate for improving accuracy. Our approach leverages a deductive system that includes SQL and Datalog, and we present an implementation in this system which is currently being tested in classroom, showing its advantages and differences with respect to other approaches, as well as some performance data.

KEYWORDS: Constraint Logic Programming, SQL, Semantic Checking, Datalog Educational System

1 Introduction

Aiding programmers with both syntax and type checking at compile-time obviously improves productivity. In the realms of SQL, current systems (both proprietary and open-source) typically lack of more advanced techniques such as, in particular, the semantic analysis of statements. After the syntax checking stage, such an analysis should point out possible incorrect statements (e.g., missing or incorrect tuples in the actual outcome). However, in this paper, we avoid actual execution of statements as done in other approaches (assessment tools, test case generation, data provenance. . . (Javid et al. 2012)), and we target at the compile-time stage instead.

There are some indicators of bad statement design which can be used to raise semantic warnings. In particular, we focus on SQL semantic errors as described in (Brass and Goldberg 2006) that can be caught independently of the database instance. There are many possible errors and, among them, the following are included: inconsistent, tautological and simplifiable conditions, uncorrelated relations in joins, unused tuple variables, constant output columns, duplicate output columns, unnecessary general comparison operators, and several others.

[*] Work partially funded by the Spanish Ministry of Economy and Competitiveness, under the grant TIN2017-86217-R (CAVI-ART-2), and by the Madrid Regional Government, under the grant S2018/TCS-4339 (BLOQUES-CM), co-funded by EIE Funds of the European Union. Special thanks are due to the anonymous referees.

Applying such a semantic analysis to SQL is cumbersome because its syntax and semantics do not facilitate expressing program properties (Guagliardo and Libkin 2017). To ease this task we use Constraint Logic Programming (CLP) (Jaffar and Lassez 1987; Apt 2003) as a reasoning setting for SQL statements. This way, we translate an SQL statement into a constraint logic program that in particular models conditions and expressions. This CLP program is then evaluated to obtain properties of interest for the semantic analysis. For example, obtaining a failed deduction indicates an inconsistent condition.

CLP systems include different solvers for specific constraint domains such as Booleans, finite domains, reals, and rationals. Each one is an instance of the generic schema CLP($\mathcal{X}$), where $\mathcal{X}$ is a constraint domain which can be mapped to an SQL type. On the one hand, since a `WHERE` condition generally includes columns of different types, then different domains (and, therefore, constraint solvers) are expected to be involved in a single condition. On the other hand, the deduction power of each solver is limited by its constraint propagators and the kind of constraints it can deal with. For example, while a finite domain solver can handle non-linear constraints, a real solver cannot. Thus, we apply *solver cooperation* (Hofstedt 2000) to enable solver cooperation for compatible domains and interchange deductions to improve accuracy.

We have implemented our proposal in a deductive database system that includes SQL as a query language. This system (Datalog Educational System – DES (Sáenz-Pérez 2011)) is an interactive tool mainly targeted at teaching, and it is appealing for SQL learning with the aid of both syntax and semantic checking (as presented here). It has experienced more than 76K downloads and has been used in more than 50 universities around the world (cf. `des.sourceforge.net/html/facts.html`). Solving a query is via an optimized translation into a Datalog program, which is then solved by its deductive engine. Thus, we take advantage of this Datalog translation for the generation of a CLP program. To the best of our knowledge, this is the first work dealing with SQL semantic errors using CLP.

We are currently using the system for our *Databases* modules via a web interface (`desweb.fdi.ucm.es`), retrieving data to evaluate the usefulness of the semantic warnings. More than 200 student accounts have been created, and more than 3,000 logins have been registered, including 600 guest account logins. Next, the proposal is motivated by examples.

Motivating Examples Following (Brass and Goldberg 2006), a simple semantic error occurs in the following query:

```
SELECT * FROM employees WHERE dept='IT' AND dept='HR';
```

Here, the condition is trivially false due to (probably) using the wrong logical operator. Despite this, it is accepted and solved with no warning in current DBMSs.

Conditions also appear in database constraints, and may be identified as either inconsistent or tautological. Consider the following definitions, in which the constraint on the salary has the minimum and maximum values interchanged (no definite tuple could ever be inserted):

```
CREATE TABLE departments(dept VARCHAR(10) PRIMARY KEY, dname VARCHAR(20));
CREATE TABLE employees(ename VARCHAR(20), dept VARCHAR(10) REFERENCES departments,
                       salary INT CHECK salary BETWEEN 5000 AND 2000);
```

Tautological conditions can occur as in the following statement (where the intention would probably be to use `AND` instead of `OR`):

```
CREATE TABLE employees(ename VARCHAR(20), dept VARCHAR(10) REFERENCES departments,
                       salary INT CHECK salary > 2000 OR salary < 5000);
```

We can consider a more involved example including both database constraints in a table and an SQL query. On it, a table is defined for containing gas products and describing their composition as percentages, which must make a total of one hundred percent. The `SELECT` query below would be inconsistent because it is asking for a gas product with components summing more than 100%.

```
CREATE TABLE gas_products(name      VARCHAR(20) PRIMARY KEY,
                          butane    FLOAT CHECK butane    BETWEEN 0 AND 100,
                          propane   FLOAT CHECK propane   BETWEEN 0 AND 100,
                          olefins   FLOAT CHECK olefins   BETWEEN 0 AND 100,
                          diolefins FLOAT CHECK diolefins BETWEEN 0 AND 100,
                          CHECK     butane+propane+olefins+diolefins =  100);

SELECT name FROM gas_products WHERE butane>60 AND propane>50;
```

Finally, another possibility is a condition that can be simplified, which may be a symptom of a wrong condition. For example:

```
SELECT butane, propane FROM gas_products
WHERE butane-propane=10 AND butane+propane=80;
```

This is equivalent to the simple condition `butane=45 AND propane=35` because the condition represents a system of linear equations with a single solution. Then, both output columns are constants, and therefore symptoms of a wrong query. Other errors that students typically make and are also covered by this tool are presented in Subsection 4.4.

Next sections detail our approach to identify such wrong uses of SQL conditions, and consists in translating an SQL statement into a CLP program, which is evaluated for identifying inconsistency, tautology, simplifiable conditions, and constant output columns. Since we use DES, which translates SQL to Datalog, we start from this translation (Section 2) for generating the CLP program (Section 3). Section 4 presents the working system with the techniques used to identify a collection of semantic errors, together with performance data. Section 5 relates this work to other approaches and, finally, we present in Section 6 our conclusions and points for future work.

2 From SQL to Datalog

In a first stage, we take advantage of the translation from an SQL query to a semantically equivalent Datalog program. It builds upon the basic presentation in (Ullman 1988), and extended in (Sáenz-Pérez 2017) (where formal results for semantic equivalence are given). First, we specify the syntax of the language fragments we consider for both SQL and Datalog, then we describe the translation, and finally some examples are presented.

2.1 SQL Syntax

In this section, we consider a fragment of standard SQL (ISO/IEC 2016). Despite our approach supports a wider coverage of SQL than the considered here, we stick to the grammar in Figure 1 for the sake of simplicity. There, `true-typed` words stand for terminal symbols, 'c' for constants, 'r' for relations (either tables or views), and 'a' for relation

```
query ::= SELECT [ALL|DISTINCT] [TOP c ] exp,...,exp FROM rel,...,rel [ WHERE cond ] |
          query s_op [ALL|DISTINCT] query
exp   ::= c | r.a | exp m_op exp | −exp | query
rel   ::= r | query
cond  ::= exp c_op exp | NOT cond | cond l_op cond | TRUE | FALSE | exp IN query | EXISTS query
c_op  ::= > | < | = | <> | >= | <=
m_op  ::= + | - | * | /
s_op  ::= UNION | EXCEPT | INTERSECT
l_op  ::= AND | OR
```

Fig. 1. A grammar for a subset of standard SQL

attributes. In this grammar (and the one in the next subsection), we use the symbol `::=` for defining parts of the language, square brackets (`[ ]`) to delimit optional parts, and vertical bars (`|`) to separate alternative parts. We assume a type inference system for syntactically valid queries. Also, we assume that syntax comprehensions such as `E BETWEEN E1 AND E2` are re-written in their equivalent basic forms supported by the grammar. Each relation alias `Relation AS Alias` in a `FROM` clause is re-written as a reference to the alias, by adding a new relation `Alias` ← `Relation` to the database, where the symbol ← stands for relation definition.

A query can appear directly as a row-returning SQL statement, as well as in other statements of the DML (Data Manipulation Language) such as `INSERT` and `DELETE` statements. For example, in: `INSERT INTO` r *query* (where the results of *query* are inserted into the relation 'r'). DDL (Data Definition Language) statements such as `CREATE TABLE` can include predicates (following the syntax of *cond* in Figure 1) in `CHECK` constraints. Note that both conditions and expressions can include queries as it can be seen in the definition of *cond* and *exp*, respectively. The DDL statement `CREATE VIEW AS` *query* also includes a query (following the syntax of *query* in the same figure). Thus, queries and conditions occurring in any part of an SQL statement are targets for the proposed semantic analysis.

2.2 Datalog Syntax

With respect to Datalog, we consider an extended Datalog language with duplicates and metapredicates as shown in Figure 2, where *rule* stands for rules, *goal* for goals, *exp* for expressions, 'atom' for an atom (possibly containing variables and constants), the comma (',') for a conjunction, and the semi-colon (';') for a disjunction. *c_op* and *m_op* are the same as in Figure 1 excepting `<=`, which is written as `=<`. The syntax of the logic includes a universe of constant symbols, a set of variables, a set of user-defined predicates, and a set of built-in metapredicate symbols (where the prefix operator `not` stands for negation, the predicate `distinct`/1 for duplicate elimination, and `top`/2 for the first n solutions of a goal). Following Prolog syntax, variables are written starting with either an upper-case letter or an underscore, and the rest of symbols either starting with lower-case or delimited by single quotes. The first form of *rule* in the figure is also known as a fact. A Datalog database (also referred to as a program) contains facts and rules instead of relation definitions as in SQL (tables and views, respectively). We consider also a type system for Datalog for restricting valid rules with respect to type specifications.

2.3 Translation

This section describes some examples of the translation of the considered SQL and Datalog languages, extending the description in (Sáenz-Pérez 2017) with the clause `DISTINCT` and the operators `IN` and `EXISTS`. Here, we refer to the function *SQL_to_DL* as defined there (which we do not reproduce it here). It takes a relation name and an SQL query defining a relation as input, and returns a multiset of Datalog rules providing the same meaning as the SQL relation for the corresponding predicate with the same name as the relation. For a query in the top-level, we assign a relation name (`answer`) to build the outcome. From here on, set-related operators and symbols refer to multisets because SQL relations can contain duplicates.

An SQL query is preprocessed before passing it to the translation function:

- If the keyword `DISTINCT` is specified in a query Q defining a relation r, this query is re-written as follows, where a fresh relation r' is introduced, and the notation $Q[X/Y]$ means a syntactic replacement of X by Y in Q:

  $r \leftarrow$ `SELECT DISTINCT * FROM` r' $\qquad r' \leftarrow Q[$`DISTINCT`$/$`ALL`$]$

  Note that a `SELECT` statement without a `WHERE` clause means an implicit `true` condition.
- If a set operator includes (either implicitly or explicitly) the keyword `DISTINCT` in a query $Q \equiv Q_1\ s\_op\ Q_2$ defining a relation r, then it is re-written as:

  $r \leftarrow$ `SELECT DISTINCT * FROM` r' $\qquad r' \leftarrow Q_1\ s\_op$ `ALL` Q_2

In addition, we define a function to deal with a set of SQL relation definitions which can appear as the result of SQL preprocessing:

Definition 2.1
The function *SQLs_to_DL* takes a set of SQL relation definitions as input and returns the equivalent Datalog program: $\mathit{SQLs\_to\_DL}(\{r_1 \leftarrow SQL_1, \ldots, r_n \leftarrow SQL_n\}) = \bigcup_{i=1}^{n} \mathit{SQL\_to\_DL}(r_i, SQL_i)$ □

Example 2.1
Given the following table schemas:

```
CREATE TABLE dept(id CHAR(10) PRIMARY KEY, name CHAR(20), location CHAR(20));
CREATE TABLE emp(name CHAR(20) PRIMARY KEY,
                 dept CHAR(10) REFERENCES dept(id), salary INT);
```

And a query in the top level that lists the employee names and their department names:

`answer` $\leftarrow Q$

$Q \equiv$ `SELECT emp.name, dept.name FROM emp, dept WHERE emp.dept=dept.id`

Since it is a single query definition, we apply the function *SQL_to_DL* to obtain:

rule	::=	atom | atom `:-` *goal* , ... , *goal*
goal	::=	atom | `not` atom | `distinct`(atom) | `top`(c, atom) | *goal* ; ... ; *goal* | *exp c_op exp*
exp	::=	`X` | *exp m_op exp* | $-$*exp*

Fig. 2. A grammar for an extended Datalog language.

$\mathit{SQL\_to\_DL}(\texttt{answer}, Q) = \{(\texttt{answer(X}_1\texttt{,X}_5\texttt{) :- r}_1\texttt{(X}_1\texttt{,X}_2\texttt{,X}_3\texttt{,X}_4\texttt{,X}_5\texttt{,X}_6\texttt{), true, true, X}_2\texttt{=X}_4)\} \bigcup$
$\{(\texttt{r}_1\texttt{(X}_1\texttt{,X}_2\texttt{,X}_3\texttt{,X}_4\texttt{,X}_5\texttt{,X}_6\texttt{) :- emp(X}_1\texttt{,X}_2\texttt{,X}_3\texttt{), dept(X}_4\texttt{,X}_5\texttt{,X}_6\texttt{)})\}$ □

Example 2.2
Given the same table schemas as in the previous example, a query in the top-level that lists the department names with assigned employees is `answer` $\leftarrow Q$, where:

$Q \equiv$ `SELECT dept.name FROM dept WHERE dept.id IN`
`(SELECT DISTINCT dept.name FROM emp, dept WHERE emp.dept=dept.id)`

This is re-written as `answer` $\leftarrow Q_1$ and $\texttt{r}_2 \leftarrow Q_2$, where:

$Q_1 \equiv$ `SELECT dept.name FROM dept WHERE dept.id IN SELECT DISTINCT * FROM` $\texttt{r}_2$
$Q_2 \equiv$ `SELECT dept.name FROM emp, dept WHERE emp.dept=dept.id`

Since there are two relation definitions, we use Definition 2.1:

$\mathit{SQLs\_to\_DL}(\{\texttt{answer} \leftarrow Q_1, \texttt{r}_2 \leftarrow Q_2\}) = \mathit{SQL\_to\_DL}(\texttt{answer}, Q_1) \bigcup \mathit{SQL\_to\_DL}(\texttt{r}_2, Q_2)$

Q_2 is almost identical to Q in Example 2.1; the differences are an absent argument in the projection, and different names for relations and variables. The translation of `answer` $\leftarrow Q_1$ is:

$\mathit{SQL\_to\_DL}(\texttt{answer}, Q_1) = \{(\texttt{answer(X}_2\texttt{) :- dept(X}_1\texttt{,X}_2\texttt{,X}_3\texttt{), true, X}_4\texttt{=X}_1\texttt{, r}_1\texttt{(X}_4\texttt{)})\} \bigcup \emptyset \bigcup$
$\{(\texttt{r}_1\texttt{(X}_4\texttt{) :- distinct(r}_2\texttt{(X}_4\texttt{))})\}$, and then:

$\mathit{SQLs\_to\_DL}(\{\texttt{answer} \leftarrow Q_1, \texttt{r}_2 \leftarrow Q_2\}) =$
$\{(\texttt{answer(X}_2\texttt{) :- dept(X}_1\texttt{,X}_2\texttt{,X}_3\texttt{), true, X}_4\texttt{=X}_1\texttt{, r}_1\texttt{(X}_4\texttt{)}),$
$(\texttt{r}_1\texttt{(X}_4\texttt{) :- distinct(r}_2\texttt{(X}_4\texttt{))}),$
$(\texttt{r}_2\texttt{(X}_8\texttt{) :- r}_3\texttt{(X}_5\texttt{,X}_6\texttt{,X}_7\texttt{,X}_8\texttt{,X}_9\texttt{,X}_{10}\texttt{), true, true, X}_6\texttt{=X}_8),$
$(\texttt{r}_3\texttt{(X}_5\texttt{,X}_6\texttt{,X}_7\texttt{,X}_8\texttt{,X}_9\texttt{,X}_{10}\texttt{) :- emp(X}_5\texttt{,X}_6\texttt{,X}_7\texttt{), dept(X}_8\texttt{,X}_9\texttt{,X}_{10}\texttt{)})\}$ □

In these simple examples, the generated Datalog program can be simplified by removing `true` goals and explicit variable bindings (e.g., $\texttt{X}_4\texttt{=X}_1$), and by applying substitutions (a goal `X=X` is a trivially `true` goal and thus it can also be removed). In addition, folding/unfolding techniques (Burstall and Darlington 1977; Tamaki and Sato 1984) are applied to further simplify the Datalog program generated. Here, unfolding can be applied to user predicate calls for which the predicate consists of only one clause, thus removing the predicate itself. Following Example 2.1, the translated program is simplified into the following single rule (with a substitution $[\texttt{X}_4/\texttt{X}_2]$):
$\{(\texttt{answer(X}_1\texttt{,X}_5\texttt{) :- emp(X}_1\texttt{,X}_2\texttt{,X}_3\texttt{), dept(X}_2\texttt{,X}_5\texttt{,X}_6\texttt{)})\}$.

3 From Datalog to CLP

Once the Datalog program corresponding to an SQL query is obtained, we can reason in the logical level about program properties of interest. Selecting a CLP language for expressing such properties seems to be a natural choice for dealing with unbound variables in conditions. Note that SQL conditions operate on data coming from their providers (relation instances, either table contents or the result of solving a reference to a view). However, our compile-time approach avoids inspecting such instances, thus leading to non-ground conditions in general. If all the conditions are expressed as a constraint problem, solvers can be used to infer some deductions.

For example, the SQL condition `r.a > s.b AND s.b > r.a` is translated into its Datalog equivalent `X > Y, Y > X`. Since `X` and `Y` are logic variables, the condition cannot be tested by a Datalog deductive engine without resorting to retrieve data from the database instance (this would make the condition ground for concrete data in safe rules (Ullman 1988)). However, it can be posted to a solver which could deduce an inconsistent state with the help of its constraint propagators. Similarly, a tautological condition can be identified by determining whether its complement is false, as in the example in the introduction: `salary INT CHECK salary > 2000 OR salary < 5000`. The condition would be translated into `X > 2000; X < 5000` and a constraint solver can deduce that its complement `X =< 2000, X >= 5000` is false. Such a solver can also simplify its constraint store. For example, given the following condition posed in the introduction: `butane-propane=10 AND butane+propane=80`, its translation `X-Y=10, X+Y=80` forms a conjunction of linear equations for which a numeric solver can find the single solution `X=45` and `Y=35`.

Therefore, our approach consists in translating the Datalog program into a CLP program in which Datalog conditions are replaced by CLP constraints. Moreover, since this CLP reasoning operates at compile-time, deductions are independent of database instances. Nonetheless, as each variable occurring in the CLP program has attached the domain of all possible values for its type, each base relation (table) in the Datalog program is translated with the CLP constraints corresponding to the `CHECK` constraints. This way, solving the CLP program represents an abstract solving of the original Datalog program.

Next subsection identifies both SQL data types and their corresponding constraint domains. Since there can be different solvers for compatible domains (with different deduction capabilities), in Subsection 3.2 we show how they interact to improve deductions via cooperation.

3.1 Domains

Constraint solvers operate on specific constraint domains, which we map to compatible SQL data types. SQL standard data types include in particular exact numeric (`INTEGER`, `NUMERIC`, `DECIMAL`, `SMALLINT`), approximate numeric (`FLOAT`, `REAL`, `DOUBLE PRECISION`), Boolean (`BOOL`), character string (`CHAR`, `VARCHAR`), and datetime types (`DATE`, `TIME`, `TIMESTAMP`).

Constraint domains are instances of the generic schema CLP($\mathcal{X}$) (Jaffar and Lassez 1987; Apt 2003), where $\mathcal{X}$ is a constraint domain. Typically, the following domains can be found in existing implementations: $\mathcal{FD}$ (finite domain of integers, with both linear and non-linear constraints), $\mathcal{Q}$ (rational numbers with linear constraints), approximate numeric ($\mathcal{R}$), and Boolean ($\mathcal{B}$).

Further constraint solvers can be developed with the aid of Constraint Handling Rules (CHR) which eases the task of implementing specific-application constraints. In particular, string solvers (which have received recently a large amount of research (Scott et al. 2013; Caballero and Ieva 2015)) could be developed over character strings, and so do solvers over datetime types.

3.2 Solver Cooperation

Solver cooperation (Correas et al. 2018; Estévez-Martín et al. 2009; Hofstedt 2000; Monfroy and Castro 2004) is a technique enabling solver interaction with the aim to early prune the search space during solving. This is possible because different solvers prune the search space in different ways. For example, on the one hand, given the constraints `X+Y=2` and `X-Y=0`, the propagators of the CLP($\mathcal{FD}$) solver are typically not able to solve the linear system, though the CLP($\mathcal{Q}$) does. On the other hand, considering the constraints `X*X=4`, `X>0` and `X<4`, the CLP($\mathcal{FD}$) solver is able to solve the non-linear problem, whereas CLP($\mathcal{Q}$) does not. Thus, provided that CLP systems enjoy different solvers, we take advantage of them and make them to cooperate.

Logic programming (LP) systems include the Herbrand domain $\mathcal{H}$ that supports computations with symbolic equality and disequality constraints over values of any type. In the CLP setting, the domain $\mathcal{H}$ can directly cooperate with other solvers wherever each variable is attached to a single CLP solver (in order to prevent domain clash).

Each condition occurring in the Datalog program is translated into a constraint term which includes the target domain type. With this indication, the constraint is sent to the corresponding solver(s) at CLP evaluation time. If a single solver $\mathcal{X}$ is available for a given domain type, then the constraint operates on the same logic variables occurring in the translated CLP program (i.e., a direct cooperation of $\mathcal{H}$ with $\mathcal{X}$). Otherwise, when more than a solver is available (on several domains $\mathcal{X}_i$), a copy of each variable in the constraint is created for each solver, and bridge constraints are imposed to make possible the bidirectional communication between them.

Figure 3 illustrates our approach to solver cooperation (which differs from (Estévez-Martín et al. 2009) since we do not take projections into account). A bridge constraint is denoted as $\mathtt{X}^{\mathcal{X}_1}$ $\mathtt{\#==}_{\mathcal{X}_1,\mathcal{X}_2}$ $\mathtt{X}^{\mathcal{X}_2}$, which relates two variables $\mathtt{X}^{\mathcal{X}_1}$ and $\mathtt{X}^{\mathcal{X}_2}$ in the domains $\mathcal{X}_1$ and $\mathcal{X}_2$, respectively. In the figure, the constraint $\mathrm{ctr}(\mathtt{X}_1^{\mathcal{H}}, \ldots, \mathtt{X}_n^{\mathcal{H}})$ operating on the domain $\mathcal{H}$ has n variables $\mathtt{X}_i^{\mathcal{H}}$ $(1 \le i \le n)$, and there are m compatible solvers for which m equivalent $\mathcal{X}_j$ constraints $\mathrm{ctr}(\mathtt{X}_1^{\mathcal{X}_j}, \ldots, \mathtt{X}_n^{\mathcal{X}_j})$ are posted to them. For each solver $\mathcal{X}_j$, new $\mathtt{X}_i^{\mathcal{X}_j}$ variables are created, which form the equivalent $\mathcal{X}_j$ constraint. Then, $n \cdot m$ bridge constraints are created of the form $\mathtt{X}_i^{\mathcal{H}}$ $\mathtt{\#==}_{\mathcal{H},\mathcal{X}_j}$ $\mathtt{X}_i^{\mathcal{X}_j}$ $(1 \le i \le n,\ 1 \le j \le m)$, relating each pair of variables $\mathtt{X}_i^{\mathcal{H}}$ and $\mathtt{X}_i^{\mathcal{X}_j}$ once for each domain $\mathcal{X}_j$.

For example, let us consider a condition $\mathtt{X}_1$ `>` $\mathtt{X}_2$ on the domain $\mathcal{H}$, and the compatible domains CLP($\mathcal{Q}$) and CLP($\mathcal{FD}$). The equivalent constraints in CLP($\mathcal{Q}$) and CLP($\mathcal{FD}$) are respectively `clpq:`$\{\mathtt{X}_1^{\mathcal{Q}}$ `>` $\mathtt{X}_2^{\mathcal{Q}}\}$, and $\mathtt{X}_1^{\mathcal{FD}}$ `#>` $\mathtt{X}_2^{\mathcal{FD}}$ (following the concrete syntax of Prolog systems such as SICStus Prolog and SWI-Prolog, where `clpq:`$\{C\}$ is the way to post the constraint C to the CLP($\mathcal{Q}$) solver, and `#>` is the $\mathcal{FD}$ operator corresponding to `>`). Four bridges are built in this case: $\mathtt{X}_1^{\mathcal{H}}$ $\mathtt{\#==}_{\mathcal{H},\mathcal{Q}}$ $\mathtt{X}_1^{\mathcal{Q}}$, $\mathtt{X}_2^{\mathcal{H}}$ $\mathtt{\#==}_{\mathcal{H},\mathcal{Q}}$ $\mathtt{X}_2^{\mathcal{Q}}$, $\mathtt{X}_1^{\mathcal{H}}$ $\mathtt{\#==}_{\mathcal{H},\mathcal{FD}}$ $\mathtt{X}_1^{\mathcal{FD}}$, and $\mathtt{X}_2^{\mathcal{H}}$ $\mathtt{\#==}_{\mathcal{H},\mathcal{FD}}$ $\mathtt{X}_2^{\mathcal{FD}}$.

3.3 *Translation*

In this subsection, we show the translation of a Datalog program Π_{DL} into a CLP program Π_{CLP} such that Π_{CLP} represents Π_{DL}. We say that Π_{CLP} represents Π_{DL} if the meaning

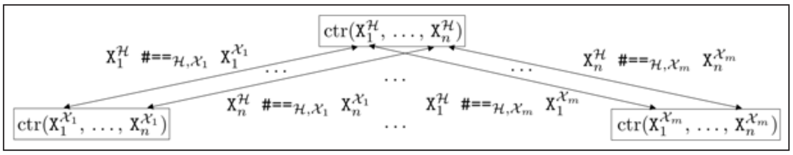
$\mathrm{ctr}(\mathtt{X}_1^{\mathcal{H}}, \ldots, \mathtt{X}_n^{\mathcal{H}})$
$\mathtt{X}_1^{\mathcal{H}}$ $\mathtt{\#==}_{\mathcal{H},\mathcal{X}_1}$ $\mathtt{X}_1^{\mathcal{X}_1}$
$\mathtt{X}_n^{\mathcal{H}}$ $\mathtt{\#==}_{\mathcal{H},\mathcal{X}_m}$ $\mathtt{X}_n^{\mathcal{X}_m}$
$\mathtt{X}_n^{\mathcal{H}}$ $\mathtt{\#==}_{\mathcal{H},\mathcal{X}_1}$ $\mathtt{X}_n^{\mathcal{X}_1}$
$\mathtt{X}_1^{\mathcal{H}}$ $\mathtt{\#==}_{\mathcal{H},\mathcal{X}_m}$ $\mathtt{X}_1^{\mathcal{X}_m}$
$\mathrm{ctr}(\mathtt{X}_1^{\mathcal{X}_1}, \ldots, \mathtt{X}_n^{\mathcal{X}_1})$
$\mathrm{ctr}(\mathtt{X}_1^{\mathcal{X}_m}, \ldots, \mathtt{X}_n^{\mathcal{X}_m})$

Fig. 3. Solver Cooperation

of Π_{DL} is included in the meaning of Π_{CLP} for any of its instance base relations. For a Datalog program Π_{DL}, its meaning (denoted as $[\![\Pi_{DL}]\!]$) is the set of ground facts inferred for each relation. For a CLP program, its meaning (denoted as $[\![\Pi_{CLP}]\!]$) is built from the set of all the (possibly non-ground) facts inferred for each relation: for each non-ground fact, all the type-compatible values constrained to the answer constraints are used to build the ground facts. Thus, Π_{CLP} represents Π_{DL} if $[\![\Pi_{DL}]\!] \subseteq [\![\Pi_{CLP}]\!]$.

For example, the meaning of the Datalog program Π_{DL} = {`r(X):-X=1;X=2`} is {`r(1), r(2)`} for a relation `r` that has integer type for its single argument. The meaning of the CLP program (omitting domain annotations) Π^1_{CLP} = {`r(X):-X>0,X<3`} is the same, provided the same integer type. Note that, whereas non-recursive Datalog enjoys finite meanings (for finite relations), CLP can have infinite meanings as, e.g., Π^2_{CLP} ={`r(X):-X>0`}, whose meaning would be {`r(1), r(2), r(3), ...`}. Both Π^1_{CLP} and Π^2_{CLP} represent the meaning of Π_{DL}, but the first one does it with a much better precision than the second one. The program Π^3_{CLP} = {`r(X):-X>0,X<2`} ($[\![\Pi^3_{CLP}]\!]$ = {`r(1)`}) does not represent Π_{DL}.

If a Datalog rule contains a call to a base relation (representing an SQL table), the translated CLP program omits that call to keep our approach instance-independent. For example, the program {`r(X):-t(X), X>17`}, with a call to the base relation `t`, is translated into the CLP program {`r(X):-true, X>17`}. Moreover, if the table has `CHECK` constraints, they are also added to the rule. Assuming that the declaration for this table is `CREATE TABLE t(a INT CHECK a>=0 AND a<=100)`, then the translation of the rule becomes: `r(X):-true, X>=0, X=<100, X>17.`

Next definitions formalize this translation from Datalog rules into CLP rules:

Definition 3.1
The function *DL_to_CLP* takes a Datalog rule as input and returns a CLP rule.

DL_to_CLP((`head :- goal`$_1$`, ..., goal`$_n$)) = (`head :- goal`$'_1$ `, ..., goal`$'_n$)
where *DLGOAL_to_CLP*(`goal`$_i$) = `goal`$'_i$ □

Definition 3.2
The function *DLGOAL_to_CLP* takes a Datalog goal as input and returns a CLP goal.

DLGOAL_to_CLP(`rel`) = `ctrs`
where `rel` is a base relation, and `ctrs` is the conjunction of user-defined constraints for `rel`

DLGOAL_to_CLP((`goal`$_1$ `, goal`$_2$)) = (`goal`$'_1$ `, goal`$'_2$)
where *DLGOAL_to_CLP*(`goal`$_i$) = `goal`$'_i$

DLGOAL_to_CLP(*meta*) = `goal`$'$
where *meta* is either `top(`n`,goal)` or `distinct(goal)`, and *DLGOAL_to_CLP*(`goal`) = `goal`$'$

DLGOAL_to_CLP(`not(goal)`) = `true`

DLGOAL_to_CLP(exp_1 op exp_2) = `ctr(`exp_1 op exp_2`,` $type$`)`
where *op* is a comparison operator, and *type* is the type of the expression

The constraints and types for a given user-defined relation are taken from its metadata. Relation types are used to annotate each logic variable in the program with its corresponding type. □

Note that a negated goal resulting from the translation of a relation defined by a statement such as `EXCEPT` restricts the meaning of the relation. Here, we leave out this restriction because we do not deal with table instances. Thus, the goal translation becomes simply `true`.

Proposition 1
The translation of a Datalog program Π_{DL} into a CLP program Π_{CLP} is a correct abstraction, i.e., $[\![\Pi_{DL}]\!] \subseteq [\![\mathtt{DL\_to\_CLP}(\Pi_{DL})]\!]$. □

The proof of this proposition is straightforward by checking that no case of Definition 3.2 removes solutions.

We assume a compatible mapping between SQL types and Datalog types. From here on, we consider a Datalog type system consisting of the data types `string`, `integer` and `float`.

Example 3.1
Let us consider again the example of gas products presented in the introduction. The result of the translation of the first SQL query into Datalog, followed by a simplification is:

```
{ (answer(N) :- gas_products(N,B,P,O,D), B>60, P>50) }
```

Applying *DL_to_CLP* to this singleton, we get:

```
answer(N):-ctr(B>=0,float), ctr(B=<100,float), ctr(P>=0,float), ctr(P=<100,float),
           ctr(O>=0,float), ctr(O=<100,float), ctr(D>=0,float), ctr(D=<100,float),
           ctr(B+P+O+D=100,float), ctr(B>60,float), ctr(P>50,float).
```

The first 9 constraints correspond to the `CHECK` constraints in the `CREATE TABLE` statement, whereas the last 2 constraints correspond to the conjunctive condition in the SQL query. All the constraints have been annotated with the corresponding types declared for the table. □

3.4 Reasoning about Conditions

There are two cases to be considered: First, a CLP program resulting from the translation of an SQL query defining an n-ary relation r. In this case, the goal to test whether the query is consistent, inconsistent or simplifiable is $r(X_1, \ldots, X_n)$. Second, a CLP program resulting from the translation of an SQL condition c, as those occurring in `CHECK` constraints for a given relation r'. In this case, we build an SQL query of the form `SELECT * FROM` r' `WHERE` c defining a fresh relation r, and we refer back to the first case. In addition, we assume a function $solve(\phi, \Pi_{CLP})$ that takes a goal ϕ to be solved in the context of a logic program Π_{CLP} and returns either a success substitution or failure. This function represents the abstraction of the original Datalog program (next section briefly describes its implementation). Only deterministic goals are considered in the analysis.

Proposition 2
Given $\mathtt{DL\_to\_CLP}(\Pi_{DL}) = \Pi_{CLP}$, if $solve(\phi, \Pi_{CLP}) = \bot$, then $solve(\phi, \Pi_{DL}) = \bot$. □

This proposition follows Proposition 1 and states that if solving a goal ϕ for the logic program Π_{CLP} that is an abstraction of another program Π_{DL} leads to failure, then solving it for Π_{DL} also leads to failure. Thus, any inconsistent condition for an SQL query can be simply found by testing whether $solve(\phi, \Pi_{CLP})$ fails. The first example in the introduction is an example of this:

$\phi =$ `answer(N,D,S)`

$\Pi_{CLP} = \{$ `(answer(N,D,S) :- ctr(D='IT',string), ctr(D='HR',string)` $\}$

which fails, therefore identifying an inconsistent condition.

As well, a tautological condition can be found by complementing it and testing if $solve(\phi, \Pi_{CLP})$ fails. An example of this is the table creation of employees (also in the introduction). For it, the following query is built and translated into a fresh relation `r` $\leftarrow$ `SELECT * FROM employees WHERE salary <= 2000 AND salary >= 5000`:

$\phi =$ `r(N,D,S)`

$\Pi_{CLP} = \{$ `(r(N,D,S) :- ctr(S=<2000,integer), ctr(S>=5000,string)` $\}$

which also fails, therefore identifying a tautological condition. Obviously, this procedure is not applied to true conditions, as it is the case of `WHERE`-less statements.

Proposition 3

Given $DL\_to\_CLP(\Pi_{DL}) = \Pi_{CLP}$, if $solve(\phi, \Pi_{CLP}) = \sigma$, and $solve(\phi, \Pi_{DL}) = \theta$, then there exists η such that $\theta = \eta \circ \sigma$. □

This also follows Proposition 1 and states that the success substitution σ of solving ϕ for the logic program Π_{CLP} is more general than θ (the one for Π_{DL}). Thus, a simplifiable condition can be found by checking if any of the constrained variables in the program is bound after a successful CLP evaluation. The last query in the introduction is an example. Its translation is:

$\phi =$ `answer(B,P)`

$\Pi_{CLP} = \{$ `(answer(B,P) :-`
```
     ctr(B>=0,float), ctr(B=<100,float), ctr(P>=0,float), ctr(P=<100,float),
     ctr(O>=0,float), ctr(O=<100,float), ctr(D>=0,float), ctr(D=<100,float),
     ctr(B+P+O+D=100,float), ctr(B-P=10,float), ctr(B+P=80,float))
```
$\}$

By solving this, the program is instantiated to:

$\Pi_{CLP} = \{$ `(answer(45,35) :-`
```
     ctr(45>=0,float), ctr(45=<100,float), ctr(35>=0,float), ctr(35=<100,float),
     ctr(O>=0,float), ctr(O=<100,float), ctr(D>=0,float), ctr(D=<100,float),
     ctr(45+35+O+D=100,float), ctr(45-35=10,float), ctr(45+35=80,float))
```
$\}$

In this program, ground conditions can be replaced by true conditions (because it has been proven by *solve* that they succeed). Therefore, they are simplifiable and a warning can be raised.

4 System Implementation

In this section, a system implementing the proposed approach to SQL semantic error identification is described. We developed this proposal in the deductive database system DES (Datalog Educational System, des.sourceforge.net) version 6.0.

4.1 Defining Solver Cooperation

In the current implementation, there is opportunity for the cooperation of the exact numerical solvers $\mathcal{Q}$ and $\mathcal{FD}$. To implement this, we define the way for solving an integer constraint, which follows the approach described in Section 3.2. The following code excerpt shows posting an integer constraint to both solvers $\mathcal{Q}$ and $\mathcal{FD}$ (the case of a single solver is a simplification of this predicate and its description is thus omitted):

```
1. post_clp_ctr(ctr(Cond,integer),InputBridges,OutputBridges) :-
2.   Cond =.. [Op,L,R], copy_term([L,R],[FDL,FDR]), copy_term([L,R],[QL,QR]),
3.   term_variables([L,R],Xs),
4.   term_variables([LFD,RFD],XFDs), term_variables([LQ,RQ],XQs),
5.   op_fdop(Op,FDOp),
6.   CtrFD =.. [FDOp,LFD,RFD], CtrQ  =.. [Op,LQ,RQ],
7.   add_bridges(fd,Xs,XFDs,InputBridges,Bridges),
8.   add_bridges(q,Xs,XQs,Bridges,OutputBridges),
9.   catch(call(CtrFD),_,true), catch(clpq:{CtrQ},_,true).
```

This predicate has two input arguments and a third output argument. The first one is the constraint, the second one is the list of already built bridges along solving, and the third one is for the output bridges (possibly augmenting the input bridges with new ones). Line 2 identifies the condition with its left and right arguments (`L` and `R`, respectively) and makes a copy (to be sent later on) as part of the $\mathcal{Q}$ and $\mathcal{FD}$ constraints, which are built in line 6, and posted to the corresponding solvers in line 9. The CLP operator for the domain $\mathcal{Q}$ is the same as for Datalog, but its correspondence with the $\mathcal{FD}$ operator has to be found (line 5) to build a syntactically correct $\mathcal{FD}$ constraint. Lines 7–8 build the bridges between each variable $\mathtt{X}^{\mathcal{H}}$ in the original condition and its counterpart variables $\mathtt{X}^{\mathcal{Q}}$ and $\mathtt{X}^{\mathcal{FD}}$. Because of variable sharing, it may be the case that a bridge for a given variable has been previously built. All the previous bridges are stored in the input list `InputBridges` and no bridge is added by the predicate `add_bridges`/5 if already present in this list. This predicate has as input arguments the domain for which to build the bridge, the variables in the condition, the copy of these variables in the constraint domain, and the input bridges. Its last argument (`OutputBridges`) will contain the input bridges plus the new one (if eventually created). For each variable in `Xs`, it calls `add_bridge`/3, which adds a new bridge if needed:

```
% Existing bridge: just retrieve bindings
add_bridge(bridge(D,X,Y),Bridges,Bridges) :- bridge_in(bridge(D,X,Y),Bridges), !.
% New bridge: add it to the output list
add_bridge(bridge(D,X,Y),Bridges,[bridge(D,X,Y)|Bridges]) :-
  add_domain_binding_daemon(D,X,Y).
```

Here, a bridge has the form `bridge(D,X,Y)`, where `D` is the domain, `X` is the variable in the domain $\mathcal{H}$, and `Y` is the variable in the domain `D` (this term corresponds to the previous notation $\mathtt{X}^{\mathcal{H}}\ \mathtt{\#==}_{\mathcal{H},\mathcal{D}}\ \mathtt{Y}^{\mathcal{D}}$). The predicate `bridge_in`/2 checks if the input bridge is already built and, if so, it retrieves the bindings for both variables. Otherwise, a new bridge is created:

```
bridge_in(bridge(D,X,Y),[bridge(D,BX,Y)|_Bri]) :- var(BX), X==BX, !.
bridge_in(bridge(D,X,Y),[_|Bri]) :- bridge_in(bridge(D,X,Y),Bri).
```

Finally, the predicate `add_domain_binding_daemon` creates a daemon (which suspends the goal in its second argument until its first argument becomes ground). It is activated by grounding either `X` in $\mathcal{H}$ or `Y` in `D`:

```
add_domain_binding_daemon(fd,X,FD):- freeze(X,FD#=X), freeze(FD,X=FD).
add_domain_binding_daemon(q,X,Q):- freeze(X,clpq:{Q=X}), freeze(Q,q_to_int(Q,X)).
```

The call to the predicate `q_to_int` converts compatible numbers between rationals and integers.

The function $solve(\phi, \Pi_{CLP})$ has been implemented with a CLP metainterpreter written in Prolog. The predicate for to this function is `clp_evaluation(Goal, Program, InputBridges, OutputBridges)`, where `Goal` is the argument corresponding to ϕ, and `Program` corresponds to Π_{CLP}. The arguments `InputBridges` and `OutputBridges` are added to keep track of the bridges being created during solving. The sketch of this predicate is similar to a Prolog metainterpreter (Sterling and Shapiro 1994), but when a constraint term is identified as a goal, it calls the predicate `post_clp_ctr`/3.

4.2 A System Session

A system session log for the examples in the introduction can be found at `www.fdi.ucm.es/profesor/fernan/DES/prole2018/examples.pdf`, which we omit here for the sake of space. In addition to these examples, note that our approach is applied to conditions as complex as needed, including subqueries. For example, subqueries in expressions are allowed:

```
DES> SELECT (SELECT ename FROM employees WHERE salary BETWEEN 5000 AND 1000)
     FROM departaments WHERE dname='Human resources';
Warning: Inconsistent condition.
Warning: Missing join condition for [departments,employees].
```

The next condition includes a subquery in the `WHERE` clause and is inconsistent:

```
DES> SELECT ename FROM employees
     WHERE salary<1000 AND salary>(SELECT salary FROM employees WHERE salary>2000);
Warning: Inconsistent condition.
```

Note that inconsistency in this case is due to the combination of the conditions of both the root query and its subquery. The translations are:

Π_{DL}=$\{$(`answer(A) :- employees(A,_B,C), employees(_D,_E,F), F>2000, C>F, C<1000`)$\}$

Π_{CLP}=$\{$(`answer(A) :- ctr(F>2000,integer), ctr(C>F,integer), ctr(C<1000,integer)`)$\}$
where it can be seen that the last three constraints cannot be fulfilled.

4.3 Performance

This section describes the performance of our approach when dealing with queries of relevant size. Our experience at classroom indicates that the tool successfully handles students' queries (including long queries for solving complex SQL puzzles which were posed to outstanding students (Sáenz-Pérez 2019)). However, the tool might be used to verify very long queries which are automatically generated by other tools (e.g., handling of persistence in Object-Relational Mapping approaches such as Hibernate ORM). Thus, this section analyses the cost of an SQL query translation (i.e., the function *SQL_to_DL* to translate SQL into Datalog in Subsection 2.3, and the function *DL_to_CLP* to translate Datalog into CLP in Subsection 3.3), and CLP solving (the function *solve* in Subsection 3.4). For the experiments we have selected a database with n empty tables, each one with a column with a `CHECK` constraint over integers (so that solver cooperation is applied), and a query Q_1 consisting of $n - 1$ nested subqueries, inductively defined as:

$Q_n \equiv$ `SELECT` $\mathtt{t}_n$`.a FROM` $\mathtt{t}_n$ `WHERE` $\mathtt{t}_n$`.a>`n

$Q_i \equiv$ `SELECT` $\mathtt{t}_i$`.a FROM` $\mathtt{t}_i$ `WHERE` $\mathtt{t}_i$`.a>`i `AND t`i`.a IN (`Q_{i+1}`)`

For a given n, Q_1 is the test query involving n correlated base tables, and inheriting the constraints in each table definition. The table to the right shows times in milliseconds (got with `statistics/2` for walltime) for the analysed steps. The column 'Total' lists the total run time (this includes other tasks such as parsing and processing). In addition, for the sake of comparing this total with those of well-known relational database systems, the two final columns 'DB2' and 'Oracle' show the total run time for IBM DB2 version 11.1.0 and Oracle version 11g via an ODBC connection from the tool. As a test platform, we used a Windows 10 64-bits OS running on an Intel Xeon CPU E3-1505M v5 (4 physical cores) running at 2.8 GHz, with 16GiB RAM. We used the source distribution of DES version 6.2 (with a bit of extra code for measuring time) running on SICStus Prolog 4.4.1 64-bits. As expected, both translating Datalog to CLP, and solving the CLP program take negligible time, whereas the translation from SQL to Datalog takes a reasonable time. Note that *SQL_to_DL* includes in particular program transformations (folding/unfolding), safety checks, argument mode handling, and simplifications. Both DB2 and Oracle do not seem to scale well with this kind of queries. Other database systems at hand (MySQL 5.7.13 and MS SQL Server 2014) could not handle so many nested levels (with a limit of 64 and 18, respectively).

n	*SQL_to_DL*	*DL_to_CLP*	*solve*	Total	DB2	Oracle
10	3	0	1	65	169	111
20	11	0	1	118	340	200
30	28	0	1	196	607	631
40	63	1	2	314	2,135	1,821
50	121	1	2	510	4,278	4,425
60	205	1	3	753	7,990	9,292
70	307	2	4	1,113	15,298	18,049
80	438	1	4	1,491	28,288	30,215
90	633	2	4	2,050	48,314	50,350
100	1,077	2	7	2,639	96,139	78,292

4.4 Supported Semantic Errors

This section lists and briefly describes our approach to deal with all the supported semantic errors (identified by numbers in (Brass and Goldberg 2006)) in our implementation. The analysis incorporates the bindings produced along a successful CLP program solving.

- Error 1: Inconsistent condition. If the evaluation of the CLP program fails, a warning is issued. This can be easily extended to display the source condition corresponding to the failing constraint by annotating each constraint with its corresponding condition.
- Error 2: Unnecessary `DISTINCT`. A warning is issued if the query returns no duplicates and includes this modifier with respect to the primary keys in the involved relations.
- Error 3: Constant output column. As a consequence of CLP solving, a column can become ground.
- Error 4: Duplicated column values. Two or more columns can be assigned to the same logical variable representing its output.
- Error 5: Unused tuple variable. An unaccessed single relation in the FROM list from the root query (Error 27 captures all other cases).
- Error 6: Unnecessary join. Check if no column in a join is used in addition to its correlation, if any. Foreign keys are taken into account, otherwise, false positives might be raised.

- Error 7: Tuple variables are always identical. A warning is issued if two or more relations produce the same tuples. This is accomplished by testing if the same goal occurs more than once with the same variables.
- Error 8: Implied or tautological condition. The original Error 8 included an inconsistent condition, which is checked in Error 1 above. Checking this is based on testing whether the complement of the condition fails, meaning that the condition is trivially true.
- Error 9: Comparison with `NULL`. This is performed in the SQL syntax tree by looking for comparisons with null values.
- Error 11: Unnecessary general comparison operator. A warning is issued if `LIKE '%'` occurs, which is equivalent to `IS NOT NULL` by inspecting the SQL syntax tree. Additionally it issues a warning about trivially true (resp. false) conditions as `cte LIKE '%'` (resp. `NOT LIKE`). This might also be checked by a string solver in Error 1.
- Error 12: `LIKE` without wildcards. Again, this error is straightforwardly checked by inspecting the SQL syntax tree.
- Error 13: Unnecessarily complicated `SELECT` in `EXISTS`-subquery. Detect patterns different from `SELECT *` as the root in an existential subquery.
- Error 16: Unnecessary DISTINCT in aggregation function. A warning is issued if either `MIN` or `MAX` is used with a `DISTINCT` argument, as well as if other aggregate is used with a `DISTINCT` expression involving key columns. In both cases, the Datalog translation is inspected.
- Error 17: Unnecessary argument of `COUNT`. A warning is issued if `COUNT` is applied to an argument that cannot be null as a primary key. Metadata is used to determine non-null arguments.
- Error 27: Missing join condition. A warning is issued if two relations are not joined by a criterium. This includes Error 5 for a single unused relation.
- Error 32: Strange `HAVING`. A warning is issued if a `SELECT` with `HAVING` does not include a `GROUP BY` by inspecting the SQL syntax tree.
- Error 33: `SUM(DISTINCT ...)` or `AVG(DISTINCT ...)`. A warning is issued if duplicate elimination is included for the argument of either `SUM` or `AVG`. If included, this might not be an error, but it is suspicious because duplicates are usually relevant for these aggregates.

5 Related Work

There are many tools targeted at learning SQL focusing on the answers of queries with respect to database instances, i.e., comparing the output of queries with the output of reference queries provided by experts (e.g., SQLator (Sadiq et al. 2004), WebSQL (Allen 2000), AsseSQL (Prior 2014), and QueryViz (Gatterbauer 2011)). Other set of tools are focused on the semantic aspects of queries as SQL Tutor (Mitrovic 2012) which uses Constraint-Based Modelling (CBM) to form models of its students, enabling the automatic selection of problems based on these models. Another one is SQL-LTM (Dollinger 2010), a tutoring module relying on reference queries to which student queries are compared.

The semantic-based system `sqllint` (Brass and Goldberg 2006) is the closest approach to ours. They introduce the concept of soft keys (attributes used in practice for identifying tuples, but that can have duplicates; e.g., the name of a person) which would be useful for some of their checks. A description of the way to identify such errors is given in (Brass and Goldberg 2005), relying on *ad-hoc* consistency checks somewhat based on classical techniques (Guo et al. 1996). With respect to subqueries, it only supports `EXISTS` (no `IN`, `>= ALL`, ...) Their approach neither supports aggregates, nor `UNION`, nor `LIKE`, and nor `IS [NOT] NULL`. As types, it includes only strings and integers, and expressions are not allowed. Finally, it does not support `CHECK` constraints in table definitions. Note also that, in contrast to DES, `sqllint` is only an analyzer, not a complete SQL system with a solving engine which could be used for teaching.

6 Conclusions and Future Work

We have presented a system using constraint logic programming for the semantic analysis of SQL statements (both DML and DDL). With the aim of detecting possible misuses of syntactically correct SQL statements at compile-time, this system focuses on both metadata and statements, instead of data from tables. There have been other approaches to SQL analysis (targeted at comparing results for concrete database instances, and based on CBM techniques), but ours mainly follows the same path as `sqllint`. However, instead of using consistency techniques as in that work, we use CLP constraints and solver cooperation to develop a precise analysis, which can deal with non-linear conditions and queries as complex as needed. Reasoning at the logic level eases the development of this approach, instead of using the more cumbersome SQL formulations for consistency checking. Performance data show that the approach is practical, and well able to cope with queries that other systems cannot afford.

Despite we have successfully evaluated the tool in classroom and students have appreciated the semantic feedback, a more thorough evaluation must be done. As part of a teaching innovation project, we are currently analysing the tool with both on-line questionnaires provided to students, and logging user sessions. While questionnaires include selectable answers in a Likert scale (and also open answers to express additional specific comments), logs can be inspected to observe the reaction of students to the semantic warnings. In addition, there is ample room for future work as, for example, the development (most likely, with CHR) of specific solvers for types such as strings and dates (in addition to the already used, but simpler, domain $\mathcal{H}$). Taking into account bindings and domain pruning in negated CLP goals resulting from the translation of constructs such as `NOT IN` and `NOT EXISTS` would also increase the precision of the analysis. Also, the tool might propose simplified versions of conditions by decompiling the CLP constraint store into SQL. Other potential additions include: a Boolean type and its handling with a CLP($\mathcal{B}$) solver; taking advantage of subtypes (exact numeric types with limited range, and string subtypes with bounded size); and implementing projections (Estévez-Martín et al. 2009). Finally, the proposal in this work can be applied to the semantic analysis of Datalog queries and programs, and to other scenarios such as the verification of automatically-generated SQL queries.

References

Allen, G. N. 2000. WebSQL: An Interactive Web Tool for Teaching Structured Query Language. In *Proceedings of Americas' Conference on Information Systems (AMCIS 2000).*

Apt, K. 2003. *Principles of Constraint Programming.* Cambridge University Press, NY, USA.

Brass, S. and Goldberg, C. 2005. Proving the safety of SQL queries. In *Fifth International Conference on Quality Software (QSIC'05)*. 197–204.

Brass, S. and Goldberg, C. 2006. Semantic Errors in SQL Queries: A Quite Complete List. *The Journal of Systems and Software 79,* 5, 630–644.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *Journal of the ACM 24,* 1, 44–67.

Caballero, R. and Ieva, C. 2015. Constraint Programming Meets SQL. In *XV Jornadas sobre Programación y Lenguajes, PROLE 2015 (SISTEDES)*.

Correas, J., Estévez-Martín, S., and Sáenz-Pérez, F. 2018. Enhancing set constraint solvers with bound consistency. *Expert Systems with Applications 92*, 485 – 494.

Dollinger, R. 2010. SQL Lightweight Tutoring Module – Semantic Analysis of SQL Queries based on XML Representation and LINQ. In *Proceedings of EdMedia 2010*. Toronto, Canada, 3323–3328.

Estévez-Martín, S., Fernández, A. J., Sáenz-Pérez, F., Hortalá-González, T., Rodríguez-Artalejo, M., and del Vado Vírseda, R. 2009. On the Cooperation of the Constraint Domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ in *CFLP*. *Theory and Practice of Logic Programming 9:4*, 415–527.

Gatterbauer, W. 2011. Databases will Visualize Queries too. *Proceedings of the VLDB Endowment 4,* 12, 1498–1501.

Guagliardo, P. and Libkin, L. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proceedings of the VLDB Endowment 11,* 1, 27–39.

Guo, S., Sun, W., and Weiss, M. A. 1996. Solving satisfiability and implication problems in database systems. *ACM Transactions on Database Systems 21*, 270–293.

Hofstedt, P. 2000. Cooperating Constraint Solvers. In *Sixth International Conference on Principles and Practice of Constraint Programming – CP*, R. Dechter, Ed. LNCS, vol. 1894. Springer-Verlag.

ISO/IEC. 2016. SQL:2016 ISO/IEC 9075-1:2016 Standard.

Jaffar, J. and Lassez, J.-L. 1987. Constraint logic programming. In *Proc. of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. ACM, 111–119.

Javid, M., Embury, S., Srivastava, D., and Ari, I. 2012. *Diagnosing faults in embedded queries in database applications*. ACM, 239–244.

Mitrovic, A. 2012. Fifteen years of constraint-based tutors: what we have achieved and where we are going. *User Modeling and User-Adapted Interaction 22,* 1, 39–72.

Monfroy, E. and Castro, C. 2004. A component language for hybrid solver cooperations. In *ADVIS*. Lecture Notes in Computer Science, vol. 3261. Springer, 192–202.

Prior, J. R. 2014. AsseSQL: An Online, Browser-based SQL Skills Assessment Tool. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ITiCSE '14. ACM, New York, NY, USA, 327–327.

Sadiq, S., Orlowska, M., Sadiq, W., and Lin, J. 2004. SQLator: An Online SQL Learning Workbench. *SIGCSE Bulletin 36,* 3, 223–227.

Sáenz-Pérez, F. 2011. DES: A Deductive Database System. *Electronic Notes on Theoretical Computer Science 271*, 63–78.

Sáenz-Pérez, F. 2017. Intuitionistic Logic Programming for SQL. In *Logic-Based Program Synthesis and Transformation*, M. V. Hermenegildo and P. López-García, Eds. Springer, 293–308.

Sáenz-Pérez, F. 2019. Experiencing Intuitionistic Logic Programming in SQL Puzzles (Work In Progress). In *XIX Conference on Programming and Languages, PROLE'2019 (SISTEDES)*. In Press.

Scott, J. D., Flener, P., and Pearson, J. 2013. Bounded strings for constraint programming. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. 1036–1043.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog (2nd Ed.): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA.

Tamaki, H. and Sato, T. 1984. Unfold/fold transformations of logic programs. In *Proceedings of the International Conference on Logic Programming*. 127–138.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems*. Computer Science Press.