Experiencing Hypothetical Datalog in SQL Puzzles

Fernando Sáenz-Pérez[1]

[1]Faculty of Computer Science, Complutense University of Madrid, 28040 Madrid, Spain

Abstract

This work presents some SQL puzzles making use of the application of Hypothetical Datalog (HD) to implement an SQL system. HD provides a way of declaring SQL Common Table Expressions as used to specify local view definitions and recursive queries. We present the concepts of HD that will be employed to translate SQL queries to HD, providing its syntax, an inference system and translation rules. Finally, several SQL puzzles used during teaching SQL in a database subject are proposed.

Keywords

Hypothetical Datalog, Recursive SQL, Teaching, Puzzles

1. Introduction

SQL is usually understood as a specific-purpose, declarative programming language, intended to insert and extract data from relational databases. However, since the addition of recursive common table expressions (CTE, included for the first time in the ISO/IEC Standard SQL:1999, also known as SQL3) its nature has evolved to become a Turing-complete language. To acknowledge this, a specific class of tag systems, CTS's (Cyclic Tag Systems) were proposed in [1], and a corollary of its work was that a CTS can emulate a Turing-complete class of tag systems. Since a CTS can be implemented in standard SQL,[1] this proves that SQL is Turing complete.

This result can be used to pose more advanced problems as puzzles to students with the aim of sharpening their programming skills and promoting open-mindedness. Using puzzles to this end is a well-known technique [2], and in fact they are used all around the world as in the *Hour of Code*,[2] *Programming Praxis*,[3] *CodeKata*,[4] *¡Acepta el reto!*[5] and also specific for SQL [3].[6] Our experience and student feedback when teaching database concepts to advanced students (typically in the double degree of Computer Science and Mathematics) during last years confirms the usefulness of puzzles as an intellectual challenge posed to students (proposed as extra-curricular activities). This paper collects some of those SQL puzzles, ranging from the easier ones to the more complex and appealing ones. Though SQL can not be considered as an esoteric language [4], expressing these puzzles with this language can be viewed as an esoteric usage.

While these puzzles can be solved in most SQL systems, when teaching we use instead the Datalog Educational System (DES)[7] [5]. This system, which is capable of solving standard SQL queries, is in our view more adequate for students than others because, in particular, it provides better syntax error explanations and even semantic error warnings [6]. Moreover, students can use its SQL declarative debugger [7] for fixing semantically incorrect views. These last two features are absent in commercial SQL systems. Apart from this, DES extends the SQL language beyond the standard, including the

Datalog 2.0 2024: 5th International Workshop on the Resurgence of Datalog in Academia and Industry, October 11-14, 2024, Dallas, Texas, USA

fernan@sip.ucm.es (F. Sáenz-Pérez)

0000-0001-6075-4398 (F. Sáenz-Pérez)

[1]https://wiki.postgresql.org/index.php?title=Cyclic_Tag_System&oldid=15106

[2]https://hourofcode.com

[3]https://programmingpraxis.com

[4]http://codekata.com

[5]Face the challenge! https://www.aceptaelreto.com

[6]SQLzoo https://sqlzoo.net, HackerRank https://www.hackerrank.com/domains/sql

[7]Desktop applications for different OS's and Prolog host systems (https://des.sourceforge.io), and an on-line system DESweb (https://desweb.fdi.ucm.es) which includes the last development version are available.

possibility of making assumptions, both positive (assuming data which is not in the database) and negative (neglecting data which is already in the database) for querying the database. However, since these puzzles are targeted at database students, we restrict ourselves to uses admitted by the SQL standard and leave the extension out of the scope of this paper.

DES is built with concepts from intuitionistic logic programming (ILP) [8] dealing to a Hypothetical Datalog (HD) language, which makes possible those positive and negative assumptions. As a consequence, it can be used to express temporary relations, similar to what an SQL `WITH` clause allows [9]. Each SQL query submitted to this system is translated into an HD program and solved by its logic engine. This paper extends [9] by introducing expressions in conditions and meta-predicates which are needed to support the translation of standard SQL queries with expressions in `WHERE` clauses and aggregates as required to solve those puzzles.[8]

1.1. Motivation

SQL data-retrieval queries usually start with a `SELECT` keyword, which builds the outcome in terms of its expression list. An exception to this is the `WITH` clause, which permits local view declarations which can be used in its outcome SQL statement. These declarations are known as Common Table Expressions (CTE's), and its syntax in recursive `WITH` queries can be simplified as:

```
WITH RECURSIVE
     R1 AS SQL1, -- CTE1
     ...,
     Rn AS SQLn  -- CTEn
SQL              -- Outcome SQL
```

where each R_i is a local, temporary view name defined by the SQL query SQL_i, and can be referenced only in the context of each SQL_i and SQL, which is the query that builds the outcome. Each declaration R_i `AS` SQL_i is a CTE. Whilst a local view declaration is similar to a `CREATE VIEW` declaration, its scope is restricted as already noted, and its lifetime is the same as for SQL.

Though CTE's are used for a number of reasons (divide-and-conquer, readability, creating views for which the user has no rights, ...), we are interested in the use that enables SQL as a Turing complete language: by specifying recursive queries. As an example, the following defines natural numbers:

```
WITH RECURSIVE
  nat(n) AS (SELECT 0 UNION ALL SELECT n+1 FROM nat)
SELECT n FROM nat;
```

where the keyword `RECURSIVE` is required by the standard to specify a (potentially) recursive CTE, the first `SELECT` is the base case (simply returning the tuple `(0)` as a `FROM`-less statement), which is unionised with the inductive case (second `SELECT`). This single CTE is used in the outcome SQL (third `SELECT`) to deliver all the naturals. Query semantics can be understood as finding its fixpoint for the application of the unionised queries on an enlarging relation `nat`, which is increased with a new number in each fixpoint-search iteration. In this case, the fixpoint would be infinite, though top-N queries can limit the number of retrieved tuples.

Current SQL systems include limitations on recursive queries: mutual and non-linear recursion are not allowed, only one recursive case can be set, and duplicate elimination is not allowed in the union of the base and inductive cases. These limitations can be overcome with other database languages such as Datalog operating over the same relational data, a language capable of expressing mutual and non-linear recursion, and duplicates [10, 11]. However, local (temporary) view definitions as needed to represent the behavior of a `WITH` clause are not part of a Datalog language, which only allows for non-volatile predicates.

[8]This work was supported by the State Research Agency (AEI) of the Spanish Ministry of Science and Innovation under grant PID2019-104735RB-C42 (SAFER).

Here is where intuitionistic logic programming (ILP) can be of help because it adds *embedded implications* in the body of clauses. Intuitively, in an embedded implication $A \Rightarrow B$, the atom A is assumed during proving B [12], limiting its scope to this proof. A database language based on ILP is Hypothetical Datalog [8], which is extended in [9, 13, 14] with rules (not only atoms) in such assumptions, a feature that can be leveraged to represent those SQL local view definitions. Next section recalls this feature implemented in the DES system, which also includes essential SQL functionalities such as handling duplicates and aggregates.

2. Intuitionistic Logic Programming for SQL

The Hypothetical Datalog language in [9] implements a system based on intuitionistic logic programming. Here, its syntax and semantics are extended with conditions in expressions and meta-predicates for supporting common SQL needs: top-N queries (i.e., at most, N solutions to its goal are allowed), aggregates, and expressions in comparison predicates. Next, $vars(T)$ is the set of variables in T.

Definition 2.1 (Syntax).
$\rho := A \mid A \leftarrow \phi_1 \wedge \ldots \wedge \phi_n$
$\phi := A \mid \neg A \mid E_1 \diamond E_2 \mid \delta(A) \mid \tau(N, A) \mid \mathcal{G}(A, Xs, \phi) \mid \rho_1 \wedge \ldots \wedge \rho_n \Rightarrow \phi$
where ρ stands for rule, ϕ for goal, A for atom (possibly containing variables and constants, but no compound terms), N is a natural number, Xs is a list of variables, E_i are expressions, $\diamond$ is a comparison operator ($=, <, \leq \ldots$), symbols $\neg$, δ, τ and $\mathcal{G}$ are meta-predicates, and $vars(\rho_i)$ do not occur out of ρ_i.

Here, the symbol $\Rightarrow$ denotes the embedded implication (it builds a goal that can occur in the body of a rule), and δ is a duplicate elimination meta-predicate. The first condition (on compound terms) is needed for ensuring a finite fixpoint in absence of infinite relations, and the second one (on variables of rules ρ_i) ensures that ρ_i is not specialized by any substitution out of it along inferencing, so ρ_i can be assumed "as is" for any inference step, which is an adequate requirement for modelling local definitions in SQL `WITH` statements (however, it represents a limitation with respect to [12]).

Symbols $\neg$, δ, τ and $\mathcal{G}$ are meta-predicates, standing respectively for negation, duplicate elimination, top-N queries, and group-by (for solving aggregations as counts and summations, which can occur in expressions as other scalar functions). A group-by goal $\mathcal{G}(A, Xs, \phi)$ means that ground instances of A are grouped by the list of variables Xs (grouping criteria) and, for each group, there is at most one instantation such that it fulfils ϕ.

Next, we extend the inference system in [9] with inference rules for comparison operators, top-N and group-by goals. The well-known concepts of predicate dependency graph (PDG) and stratifiable database Δ [15] are used here (see also page 298 in [9]). Here, the PDG includes a negative arc $p \overset{\neg}{\leftarrow} q$ between the head predicate p of a rule and a predicate q in its body, such that q occurs in an argument of a meta-predicate $\neg$, τ or $\mathcal{G}$; otherwise, it includes a positive arc $p \leftarrow q$. A negative arc $p \overset{\neg}{\leftarrow} q$ imposes that q must be solved before p for avoiding non-monotonicity [15]. $str(\Delta, P)$ is the stratum corresponding to predicate P in Δ [9] such that for a positive arc $\mathtt{p} \leftarrow \mathtt{q}$, $str(\Delta, \mathtt{p}) \geq str(\Delta, \mathtt{q})$, and for a negative arc $\mathtt{p} \overset{\neg}{\leftarrow} \mathtt{q}$, $str(\Delta, \mathtt{p}) > str(\Delta, \mathtt{q})$.

$\Delta \vdash \psi$ is an *inference expression*, where ψ is an identified ground atom $id : A$. Such an id is of the form $id_1 \cdot \cdots \cdot id_n$ ($n \geq 1$) and helps data provenance for supporting duplicates. $pred(A)$ is the predicate of atom A. A *negative inference expression* is of the form $\Delta \vdash id : -A$. Negative inference expressions play a key role in the SQL extension for negative assumptions, but are out of the scope of this paper. Next definition specifies an inference system where each rule is read as: If the formulas above the line can be inferred, then the ones below the line can be inferred as well. The application of an inference rule is an inference step.

Definition 2.2 (Inference system). Given a database Δ and a set of input inference expressions $\mathcal{E}$, the inference system associated to the s-th stratum is defined as follows, where $d_s(\mathcal{E})$ is a closure operator that denotes the set of inference expressions derivable in this system:
Axioms:

- $\Delta \vdash id : A$ is an axiom for each (ground) atomic formula $id : A$ in Δ, where $str(\Delta, pred(A)) = s$.
- $\Delta \vdash c_1 \diamond c_2$ is an axiom, where $\diamond \in \{=, <, >, \leq, \geq\}$, c_i are constants, and $c_1 \diamond c_2$ holds.
- Each expression in $\mathcal{E}$ is an axiom.

Inference Rules:

- For any rule $A \leftarrow \phi_1 \wedge \ldots \wedge \phi_n$ with identifier id in Δ, where $str(\Delta, pred(A)) = s$ and for any ground substitution θ:
$$\frac{\Delta \vdash id_i : \phi_i\theta \text{ for each } i}{\Delta \vdash id' : A\theta} \quad (\textit{Clause})$$
where id' is the composition of identifiers $id \cdot id_1 \cdots id_n$.

- For any constants c_i and expressions e_i:
$$\frac{\Delta \vdash c_1 \diamond c_2}{\Delta \vdash e_1\theta \diamond e_2\theta} \quad (\textit{Expression})$$
where $\diamond \in \{=, <, >, \leq, \geq\}$, and $e_i\theta$ evaluates to c_i.

- For any atoms A, A':
$$\frac{\Delta \vdash id : A}{\Delta \vdash \alpha : \delta(A'\theta)} \quad (\textit{Duplicates})$$
where $A = A'\theta$, α is a single, unique identifier.

- For any atoms A_i, A, natural number N, and ground substitutions θ_i:
$$\frac{\Delta \vdash id_i : A_i}{\Delta \vdash \alpha_i : \tau(N, A\theta_i)} \quad (\textit{Top})$$
where $A_i = A\theta_i$, α_i are at most N single, unique identifiers.

- For any atoms A_i, A, B_j, B and ground substitutions θ_i, θ:
$$\frac{\Delta \vdash id_i : A_i \quad \Delta \vdash id_j : B_j}{\Delta \vdash \alpha_i : \mathcal{G}(A\theta_i, Xs\theta_i, B\theta_i)} \quad (\textit{Group-By})$$
where each α_i is a unique identifier for each group $Xs\theta_i$, and $A = A_i\theta$, $B = B_j\theta$.

- For any goal ϕ:
$$\frac{\Delta \cup \{\rho_1, \ldots, \rho_n\} \vdash \phi}{\Delta \vdash \rho_1 \wedge \ldots \wedge \rho_n \Rightarrow \phi} \quad (\textit{Assumption})$$

Example 1. Natural numbers can be specified in this language as follows:

$$\phi_1 \equiv (nat(0) \wedge (nat(X) \leftarrow nat(Y) \wedge X = Y + 1)) \Rightarrow nat(X)$$

Starting with $\Delta = \emptyset$, and applying the *Assumption* rule:

$$\frac{\{(nat(0)), (nat(X) \leftarrow nat(Y) \wedge X = Y + 1)\} \vdash nat(X)}{\emptyset \vdash (nat(0) \wedge (nat(X) \leftarrow nat(Y) \wedge X = Y + 1)) \Rightarrow nat(X)}$$

Δ, originally empty, has been enlarged with the two rules in the assumption (let us call it Δ'), and each one receives a unique identifier (as any other possible rule in the program). For this example, let r_1 and r_2 be the respective identifiers for $nat(0)$ and $nat(X) \leftarrow nat(Y) \wedge X = Y + 1$. In a second inference step, we can apply the *Clause* inference rule, giving: $\Delta' \vdash r_1 : nat(X)\{X \mapsto 0\} \equiv \Delta' \vdash r_1 : nat(0)$, where $\theta = \{X \mapsto 0\}$, and $\Delta' \vdash r_1 : nat(0)$ represents a first solution of the query. This can be read as: the atom $nat(0)$ identified by r_1 has been inferred for the context Δ'. Next, using r_2 in *Clause* for some id after *Assumption* yields:

$$\frac{\Delta' \vdash id : nat(Y)\theta_1 \quad \Delta' \vdash (X = Y + 1)\theta_1}{\Delta' \vdash r_2 \cdot id : nat(X)\theta_1}$$

where axioms above the line can be inferred: First, the axiom $\Delta' \vdash r_1 : nat(0)$ (with $\theta = \{Y \mapsto 0\}$) as before, and the second one because it becomes a trivial axiom $\Delta' \vdash 1 = 1$ after applying *Expression* (i.e., the evaluated $X = 0 + 1$ for $\{X \mapsto 1\}$). So $\theta_1 = \{Y \mapsto 0, X \mapsto 1\}$, and the axiom $\Delta' \vdash r_2 \cdot r_1 : nat(1)$ can be inferred. Proceeding in the same way, the following axioms are inferred with $k > 0$:

$$\Delta' \vdash \underbrace{r_2 \cdots r_2}_{k} \cdot r_1 : nat(k)$$

□

Negative information is deduced by applying the closed-world assumption to a set of inference expressions $\mathcal{E}$ (written as $cwa(\mathcal{E})$) as the union of $\mathcal{E}$ and the negative inference expression for $\Delta \vdash \phi$ such that $\Delta \vdash \phi \notin \mathcal{E}$. Thus, a stratified bottom-up construction of the *unified stratified semantics* can be specified by:

- $\mathcal{E}^0 = \emptyset$
- $\mathcal{E}^{s+1} = cwa(d_{s+1}(\mathcal{E}^s))$ for $s \geq 0$.

which builds a set of axioms $\mathcal{E}$ that provides a way to assign a meaning to a goal ϕ with:

$$solve(\phi, \mathcal{E}) = \{\Delta \vdash id : \psi \in \mathcal{E} \mid \phi\theta = \psi\}$$

where θ is a substitution and each axiom in $\mathcal{E}$ is mapped to the database Δ it was deduced for, and the inferred fact ψ is labelled with its data source id (for supporting duplicates). We use $\Delta(\mathcal{E})$ to denote the multiset of facts ψ so that $\Delta \vdash id : \psi \in \mathcal{E}$ for any id. In Example 1, there is only one stratum, so:

$$solve(\phi_1, \mathcal{E}) = \{\Delta' \vdash \underbrace{r_2 \cdots r_2}_{k} \cdot r_1 : nat(k) \mid k \geq 0\} = \mathcal{E}_{\phi_1}$$

and the outcome would be the multiset $\Delta(\mathcal{E}_{\phi_1}) = \{nat(k) \mid k \geq 0\}$.

A top-10 goal for this example would restrict the number of solutions as: $\Delta(\mathcal{E}_{\tau(10,\phi_1)}) = \{nat(k) \mid 0 \leq k \leq 9\}$, which corresponds to modifying the outcome SQL query in the `WITH` statement by `SELECT TOP 10 n FROM nat`.

The stratified construction of inference expressions captures the requirements of SQL clauses such as `EXCEPT`, `NOT IN`, and `GROUP BY`, which are sources of negative arcs in the construction of a predicate dependency graph (PDG) [16].

3. Translating SQL to Hypothetical Datalog

Here, the function *SQL_to_DL* is defined by extending Definition 6 in [9] with top-N queries, aggregates and expressions. This function takes a relation name and an SQL statement as input and returns a multiset of Hypothetical Datalog rules providing the same meaning as the SQL relation for a corresponding predicate with the same name as the relation. From here on: Without loss of expressivity, we assume that a group-by statement includes a relation name in its `FROM` clause; set-related operators and symbols refer to multisets because SQL relations can contain duplicates; each attribute is qualified by the relation name it corresponds to so that they can be uniquely identified; an empty conjunctive goal is $true$; where convenient, variables of goals are made explicit; and for the sake of conciseness, Hypothetical Datalog is simply referred to as Datalog.

Definition 3.1. The function *SQL_to_DL* takes a relation name and an SQL statement as input and returns a Datalog program as defined by cases as follows:

% Basic SELECT statement
SQL_to_DL$(r,$ `SELECT` $Exps$ `FROM` Rel `WHERE` $Cond)$ =
$\{ r(\overline{X}) \leftarrow DLRel(\overline{Y}) \wedge DLExps(\overline{X}) \wedge DLCond(\overline{Z}) \} \bigcup RelRules \bigcup ExpRules \bigcup CondRules$,
where *SQLREL_to_DL* $(Rel) = (DLRel(\overline{Y}), RelRules)$, *SQLEXPS_to_DL* $(Exps) =$

$(DLExps(\overline{X}), true, ExpRules)$, and $\texttt{SQLCOND\_to\_DL}(Cond) = (DLCond(\overline{Z}), CondRules)$. Here, each $\texttt{E}_i$ in the list $Exps$ is either an expression or an argument name present in the relation Rel with corresponding logic variable $X_i \in \overline{X}$. Rel is either a single defined relation (table or view), or a join of relations, or an SQL statement. Function $\texttt{SQLREL\_to\_DL}$ (respectively, $\texttt{SQLCOND\_to\_DL}$) takes an SQL relation (respectively, condition) and returns a goal with as many variables $\overline{Y}$ as arguments of Rel and, possibly, additional rules which result from the translation (and similarly for $\texttt{SQLEXPS\_to\_DL}(Exps)$). Variables $\overline{Z} \subseteq \overline{Y}$ come as a result of the translation of the condition $DLCond$ to a goal.

% GROUP BY statement
$\texttt{SQL\_to\_DL}(r, \texttt{SELECT}\ Exps\ \texttt{FROM}\ Rel\ \texttt{GROUP BY}\ Cols) =$
$\{ r(\overline{X}) \leftarrow DLExps \wedge \mathcal{G}(DLRel(\overline{Y}), \overline{V}, DLAgg) \} \bigcup ExpsRules \bigcup RelRules,$
where $\texttt{SQLEXPS\_to\_DL}(Exps) = (DLExps, DLAgg, ExpsRules)$, $\texttt{SQLREL\_to\_DL}(Rel) = (DLRel(\overline{Y}), RelRules)$. Here, $\overline{V} \subseteq \overline{Y}$ are the variables of the predicate $DLRel$ corresponding to the attributes $Cols$ of the relation Rel.

% Top-N
$\texttt{SQL\_to\_DL}(r, \texttt{SELECT TOP N}\ Exps\ \texttt{FROM}\ Rel\ \texttt{WHERE}\ Cond) = \{ r(\overline{X}) \leftarrow \tau(N, r'(X)) \} \bigcup Rules,$
where $\texttt{SQL\_to\_DL}(r', \texttt{SELECT}\ Exps\ \texttt{FROM}\ Rel\ \texttt{WHERE}\ Cond) = Rules$

% Union
$\texttt{SQL\_to\_DL}(r, SQL_1\ \texttt{UNION ALL}\ SQL_2) = \texttt{SQL\_to\_DL}(r, SQL_1) \bigcup \texttt{SQL\_to\_DL}(r, SQL_2)$

% Difference
$\texttt{SQL\_to\_DL}(r, SQL_1\ \texttt{EXCEPT}\ SQL_2) =$
$\{ r(\overline{X}) \leftarrow s(\overline{X}) \wedge \neg t(\overline{X}) \} \bigcup \texttt{SQL\_to\_DL}(s, SQL_1) \bigcup \texttt{SQL\_to\_DL}(t, SQL_2)$

% Intersection
$\texttt{SQL\_to\_DL}(r, SQL_1\ \texttt{INTERSECT}\ SQL_2) =$
$\{ r(\overline{X}) \leftarrow s(\overline{X}) \wedge t(\overline{X}) \} \bigcup \texttt{SQL\_to\_DL}(s, SQL_1) \bigcup \texttt{SQL\_to\_DL}(t, SQL_2)$

% WITH statement
$\texttt{SQL\_to\_DL}(r, \texttt{WITH}\ r_1\ \texttt{AS}\ SQL_1, \ldots, r_n\ \texttt{AS}\ SQL_n\ SQL) =$
$\{ r(\overline{X}) \leftarrow \bigwedge(\{\texttt{SQL\_to\_DL}(r_i, SQL_i) \mid 1 \leq i \leq n\}) \Rightarrow s(\overline{X}) \} \bigcup \texttt{SQL\_to\_DL}(s, SQL)$

where $\bigwedge(Bag)$ denotes $B_1 \wedge \cdots \wedge B_m$ ($B_i \in Bag$).

Next three definitions specify the functions $\texttt{SQLREL\_to\_DL}$, $\texttt{SQLEXPS\_to\_DL}$ and $\texttt{SQLCOND\_to\_DL}$ which are used by $\texttt{SQL\_to\_DL}$.

Definition 3.2. The function $\texttt{SQLREL\_to\_DL}$ takes an SQL relation (either a relation name, or a join of relations, or an SQL statement) as input and returns both a Datalog goal and program as defined by cases as follows:

% Extensional/Intensional relation name
$\texttt{SQLREL\_to\_DL}(RelName) = (RelName(\overline{X}), \{\})$
where $\overline{X}$ are the n variables corresponding to the n-ary relation $RelName$.

% Join of relations
$\texttt{SQLREL\_to\_DL}((Rel_1, \ldots, Rel_n)) = (DLRel(\overline{X_1}) \wedge \cdots \wedge DLRel(\overline{X_n}), RelRules_1 \cup \cdots \cup RelRules_n)$
where $\texttt{SQLREL\_to\_DL}(Rel_i) = (DLRel(\overline{X_i}), RelRules_i)$.

% SQL statement
$\texttt{SQLREL\_to\_DL}(SQL) = (RelName(\overline{X}), \texttt{SQL\_to\_DL}(RelName, SQL))$
where $\overline{X}$ are the n variables corresponding to the n-ary statement SQL, and $RelName$ is a fresh relation name.

Definition 3.3. The function *SQLEXPS_to_DL* takes a sequence of SQL expressions as input and returns a Datalog conjunctive goal for expressions, another for aggregates, and a program, as defined by cases as follows:

% Sequence of expressions
SQLEXPS_to_DL$((Exp_1, \ldots, Exp_n)) = (\bigwedge\{DLExp_i \mid 1 \leq i \leq n\}, \bigwedge\{DLAgg_i \mid 1 \leq i \leq n\},$
$ExpRules_1 \cup \cdots \cup ExpRules_n)$,
where *SQLEXPS_to_DL*$(Exp_i) = (DLExp_i, DLAgg_i, ExpRules_i)$.

% Expression
SQLEXPS_to_DL$(Exp) = ((X = DLExp) \wedge DLQs, DLAgg, ExpRules)$, where X is the variable corresponding to the expression Exp at position p in the list of SQL expressions $(Exp_1, \ldots, Exp_n)$, and $DLExp$ is the result of: 1) replacing each attribute a_i of a relation r in Exp by the respective variable X_i of the predicate r; 2) replacing each aggregate function f over an attribute b_j of the relation s by a fresh variable Y_j. For each f out of n, $DLAgg$ includes a conjunctive goal $Y_j = f(Z_j)$, where Z_j is the variable of $s(\overline{Z})$ which corresponds to the attribute b_j; and 3) replacing each query Q with alias a by a fresh variable U such that *SQL_to_DL*$(a, Q) = (QRules)$, $ExpRules$ contains $QRules$, and $DLQs$ includes a conjunctive goal $a(U)$.

Definition 3.4. The function *SQLCOND_to_DL* takes an SQL condition as input and returns a Datalog conjunctive goal and a program, as defined by cases as follows:

% Basic condition
SQLCOND_to_DL$(Exp_1\ cop\ \ Exp_2) = ((X_1\ cop'\ X_2) \bigwedge\{(X_i = DLExp_i) \wedge DLQs_i \mid 1 \leq i \leq 2\}, ExpRules_1 \cup ExpRules_2)$,
where *SQLEXPS_to_DL*$(Exp_i) = ((X_i = DLExp_i) \wedge DLQs_i, true, ExpRules_i)$, $1 \leq i \leq 2$, and *cop*′ is the Datalog corresponding comparison operator to *cop*.

% Conjunction
SQLCOND_to_DL$(C_1$ `AND` $C_2) = (C_1' \wedge C_2', CRules_1 \cup CRules_2)$
where C_i $(1 \leq i \leq 2)$ are conditions and *SQLCOND_to_DL*(C_i)=$(C_i', CRules_i)$.

% Disjunction
SQLCOND_to_DL$(C_1$ `OR` $C_2) = (\delta(d), \{d \leftarrow C_1', d \leftarrow C_2'\} \cup CRules_1 \cup CRules_2)$
where C_i $(1 \leq i \leq 2)$ are conditions, *SQLCOND_to_DL*(C_i)=$(C_i', CRules_i)$, and d is a fresh atom.

% Negated condition
SQLCOND_to_DL(`NOT` $C) = (C', CRules)$
where C'=*SQLCOND_to_DL*$(\overline{C})$=$(C', CRules)$, and $\overline{C}$ represents the logical complement of C ($\overline{X < Y} = X \geq Y$, $\overline{C_1 \wedge C_2} = \overline{C_1} \vee \overline{C_2}$, and so on).

Note that disjunction requires duplicate elimination because, otherwise, there can be more tuples in the result because a disjunction in a rule is rewritten as two rules, as the next example illustrates:

Example 2. Consider the SQL query `SELECT SQRT(2) FROM dual WHERE TRUE OR TRUE` defining a relation r. The relation `dual`, as originally defined by Oracle, contains a single row and a single column, and it was included as a predefined relation to allow for calculations such as this example. DES includes this as a propositional relation instead (there is no need for an argument because it is not intended to be used). Translating this query using Definition 3.1-*Basic SELECT statement*, where $Rel =$ `dual`, $Exps =$ `SQRT(2)` and $Cond =$ `TRUE OR TRUE` results in:
SQL_to_DL(r, `SELECT SQRT(2) FROM dual WHERE TRUE OR TRUE`) =
$\{ r(\overline{X}) \leftarrow DLRel \wedge DLExps(\overline{X}) \wedge DLCond \} \bigcup RelRules \bigcup ExpRules \bigcup CondRules =$
$\{ r(X) \leftarrow dual \wedge X = sqrt(2) \wedge \delta(d), d \leftarrow true, d \leftarrow true \}$
(rules with respective identifiers r_1, r_2 and r_3), because following Definition 3.2-*Extensional/Intensional relation name*: *SQLREL_to_DL*(`dual`) $= (DLRel, RelRules) = (dual, \{\})$, following Definition 3.3-*Expression*: *SQLEXPS_to_DL*(`SQRT(2)`) $= (DLExp, DLAgg, ExpsRules) = ((X =$

$sqrt(2)), true, \{\})$, and following Definition 3.4-*Disjunction*: *SQLCOND_to_DL*(TRUE OR TRUE) $= (DLCond, CondRules) = (\delta(d), \{d \leftarrow true, d \leftarrow true\})$. Thus, $solve(r(X), \mathcal{E}) = \{\Delta \vdash r_1 \cdot \alpha : r(1.4142)\}$ (where α is the fresh identifier introduced by Definition 2.2-*Duplicates*). Without duplicate elimination (goal d instead of $\delta(d)$ in rule r_1): $solve(r(X), \mathcal{E}) = \{\Delta \vdash r_1 \cdot r_2 : r(1.4142), \Delta \vdash r_1 \cdot r_3 : r(1.4142)\}$, which is not the expected outcome for the SQL query.

The actual system applies other techniques to simplify the translated program, such as partial evaluation and folding/unfolding [17], which are not presented here. In this case, the resulting translation is simply $\{r(1.4142)\}$, as it can be checked in the system by displaying compilations with the command `/show_compilations on`. Observe also that the system emits the warning for this example: `[Sem] Tautological condition: (true OR true)`, which is one of the results of its semantic analysis [6], indicating a superfluous condition. □

Next, the translation of a recursive `WITH` statement is illustrated.

Example 3. Consider the SQL query in Subsection 1.1 defining natural numbers. Translating this query follows Definition 3.1-*WITH statement*, where $r_1 =$ `nat`, *SQL*$_1$ = `SELECT 0 UNION ALL SELECT n+1 FROM nat`, and *SQL* = `SELECT n FROM nat`:
SQL_to_DL(*r*, `WITH RECURSIVE nat(n) AS (SELECT 0 UNION ALL SELECT n+1 FROM nat) SELECT n FROM nat`) = $\{ r(\overline{X}) \leftarrow \bigwedge(\{$*SQL_to_DL*$(r_i,$ *SQL*$_i) \mid 1 \leq i \leq n\}) \Rightarrow s(\overline{X}) \} \bigcup$ *SQL_to_DL*$(s,$ *SQL*$) = \{(r(X) \leftarrow ((nat(X_1) \leftarrow dual \wedge X_1 = 0 \wedge true) \wedge (nat(X_2) \leftarrow nat(Y) \wedge X_2 = Y + 1 \wedge true) \Rightarrow s(X))), (s(X_3) \leftarrow nat(Z) \wedge X_3 = Z \wedge true)\}$ because, by Definition 3.1-*Union*: *SQL_to_DL*$(nat,$ `SELECT 0 UNION ALL SELECT n+1 FROM nat`) = *SQL_to_DL*$(nat,$ `SELECT 0`$) \cup$ *SQL_to_DL*$(nat,$ `SELECT n+1 FROM nat`) and, by Definition 3.1-*Basic SQL statement*, where `SELECT 0` is an abbreviation for `SELECT 0 FROM dual WHERE TRUE`, the following is get by performing similar steps to Example 2: *SQL_to_DL*$(nat,$ `SELECT 0`$) = \{ nat(X_1) \leftarrow dual \wedge X_1 = 0 \wedge true \}$, and applying again the same definition to: *SQL_to_DL*$(nat,$ `SELECT n+1 FROM nat`$) = \{ nat(X_2) \leftarrow nat(Y) \wedge X_2 = Y + 1 \wedge true \}$, and: *SQL_to_DL*$(s,$ `SELECT n FROM nat`$) = \{ s(X_3) \leftarrow nat(Z) \wedge X_3 = Z \wedge true \}$.

Further simplification and unfolding steps done in the system results in a single rule:
$\{(r(X) \leftarrow (nat(0)) \wedge (nat(Y) \leftarrow nat(Z) \wedge Y = Z + 1) \Rightarrow nat(X))\}$. □

[9] provides a theorem for the semantic equivalence of an SQL relation and its counterpart Datalog translation for a subset of the language presented here. An SQL relation is translated into an extended relational algebra relation (ERA) definition (with *SQL_to_ERA*), for which an interpretation for a computed answer in terms of the ERA operators is provided. The Datalog program is interpreted with the unified stratified semantics, which provides the set of axioms $\mathcal{E}$ for the original database Δ augmented with the translation of the query Q to this Datalog program. So, the proof inductively proceeds by strata, and for each stratum by cases, showing the semantic equivalence of the translation from the SQL query Q into an ERA expression for a relation r (with *SQL_to_ERA*), and the translation of Q into a Datalog program (with *SQL_to_DL*).

4. SQL Puzzles

This section provides several SQL puzzles[9] provided to students and solved in DES, mainly with its on-line interface DESweb. We use the concrete SQL syntax that DES supports, allowing for omitting the verbose `RECURSIVE` modifier which, though promoted and required by the standard for recursive queries, it seems to be an unneeded writing burden (a simple program analysis can classify recursive predicates and apply both specific compilations and optimizations). Also, `UNION` can be used instead of `UNION ALL` if duplicates are not required. Even when it might seem that discarding duplicates results in a solving overhead, it turns out to be the opposite for recursive queries [18]. For each puzzle, its problem statement is given, together with an equivalent Datalog formulation, and outcome. A selection of other puzzles can be found in Appendix A.

[9] At the time of posing these puzzles and up to the best of our knowledge, most of these puzzles were novel in SQL (we only found a statement for primes), and solutions to them are reported here for the first time, both in SQL and Datalog.

Here, we use the concrete syntax of DES, where `:-` stands for the neck of a rule, the comma (`,`) for goal conjunction, `=>` for embedded implication, `/\` for rule assumptions, `distinct(`A`)` for $\delta(A)$, `top(`N`,`A`)` for $\tau(N, A)$, and `group_by(`A`,`Xs`,`ϕ`)` for $\mathcal{G}(A, Xs, \phi)$. Relation attributes are not explicitly qualified with the relation name unless necessary.

4.1. Euler Number

Problem Statement: Compute the Euler number e as a Taylor series:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots = \sum_{i \geq 0} \frac{1}{i!}$$

As stop condition implicitly use the system precision for numbers.
A Solution: A straightforward solution is to define a CTE for the factorials with a schema of two arguments: the index n (`n`) and the value of $n!$ (`factorial`). Its base case is $(0, 1)$, and the inductive case just computes (`n` + 1, `n`+1`*factorial`), given an already-computed tuple (`n`, `factorial`). Then, define each term in the series as another CTE, say `taylor`, with a single column (`euler`) with each term in the series, computed as 1/`factorial` from the CTE `factorials`:

```
WITH
  factorials(n, factorial) AS
    (SELECT 0, 1 UNION ALL SELECT n+1, (n+1)*factorial FROM factorials)
  WITH
    taylor(euler) AS
      (SELECT 1/factorial FROM factorials)
  SELECT SUM(euler) FROM (SELECT TOP 18 euler FROM taylor);
```

Note a couple of things: First, there are two nested `WITH` statements (`factorials` is only known in the context of the subsequent `WITH` statement. Alternatively, two consecutive CTE's are also possible. Second, the outcome SQL needs a top-N clause to limit recursion. Up to now, main RDBMS's as PostgreSQL, DB2, SQL Server, and Oracle do not admit nested `WITH` statements.
Datalog Formulation: Next, it is shown the automatically-generated core Datalog[10] query and program corresponding to the SQL statement. The program is asserted before solving the query and retracted afterwards.

```
factorials(0,1) /\ (factorials(B,C) :- factorials(D,E), B=D+1, C=(D+1)*E)
=> answer_1_6(A).

answer_1_6(A) :- (taylor(B) :- factorials(_C,D), B=1/D) => answer_1_8(A).

answer_1_8(A) :- group_by(answer_1_10(B),[],A=sum(B)).

answer_1_10(A) :- top(18,taylor(B)), B=A.
```

This translation folds some relations for the deductive engine to be able to solve the query. Instead, a user would alternative write a query like this:

```
factorials(0,1) /\ ( factorials(D,E) :- factorials(F,G), D=F+1,  E=(F+1)*G )
=> (taylor(B) :- factorials(_C,D), B=1/D)
=> group_by(top(18,taylor(B)),[],A=sum(B)).
```

An alternative solution is to define a CTE with an argument for the index n of the n-th term in the series, the value of the term ($term$), and the current value of the running summation (val). Thus, the base case is $n = 0$, $term = 1$, and $val = 1$, and the inductive case computes the next term for presolved cases delivering $n + 1$, $term/(n + 1)$ (which equates to divide the n-th term by $n + 1$), and accumulating in val the $n + 1$-th approximation. The stop condition should test if the last term is strictly greater than 0, since otherwise this means that the system numeric precision is met.

[10] Core Datalog is the language directly processed by the deductive engine. (User) Datalog is the language that the user can use at the top-level, and that enables high-level formulations for expressions and some metapredicates.

```
WITH
  re(n,term,val) AS
    (SELECT 0, 1.0, 1.0  UNION
     SELECT n+1, term/(n+1), val+term/(n+1) FROM re WHERE term/(n+1) > 0)
  SELECT MAX(val) AS euler FROM re;
```

Datalog Formulation:

```
re(0,1,1) /\ ( re(B,C,D) :- re(E,F,G), H=F/(E+1), H>0, B=E+1, C=F/(E+1), D=G+F/(E+1) )
=> group_by(re(_B,_C,D), [], A=max(D)).
```

By submitting this user Datalog query at the top-level, the system compiles it to an almost equal core Datalog query and program as shown above for the SQL case. Such compilations can be inspected with the command `/show_compilations on`.
Outcome: The query returns in the SICStus implementation:

```
answer(euler:float) ->
{ answer(2.7182818284590455) }
```

Obviously, the precision of the approximation to the Euler number depends on the underlying system arithmetics. Here, the last digit should read 2 instead of 5. This outcome is available by submitting the command `display_answer on`.

4.2. All Time Greatest Hits

Problem Statement: We are given the table `hits(theme string, copies int)`,[11] holding song titles and number of sold copies for each title. Write a recursive SQL query to assign the rank of each song in terms of its sold copies (number 1 is for the most sold song, number 2 for the next and so on). If there are two or more songs with the same number of copies, they will receive the same rank number.
A Solution: An idea to solve this puzzle is to build pairs with an integer representing the rank and the number of copies for hits such that there no other hits with a higher number of copies. Then, return the maximum rank for each group of hits with the same number of copies, adding to the output the song titles (known from the table hits and their number of copies).

The base case in the recursive part is rank 1 for all hits. The inductive case `rec` is the next rank in `rec` such that the copies in hits is strictly less than the copies in `rec`, i.e., once all copies receive a rank 1, the rank 2 is received to all copies but the one with the maximum number of copies, and so on. Its SQL formulation is as follows:

```
WITH rec(ranking, copies) AS
  (SELECT 1, copies FROM hits
   UNION
   SELECT ranking+1, hits.copies
   FROM hits, rec
   WHERE hits.copies < rec.copies)
SELECT ranking, theme, hits.copies
FROM hits, (SELECT MAX(ranking) AS ranking, copies FROM rec GROUP BY copies) m
WHERE m.copies = hits.copies;
```

Datalog Formulation: Observe that a Datalog equivalent query is neater than its SQL counterpart:

```
( rec(1,D) :- hits(_E,D) ) /\ ( rec(F,G) :- hits(_H,G), rec(I,J), G<J, F=I+1 )
=> hits(B,C), group_by(rec(D,C), [C], A=max(D)).
```

Outcome:

```
answer(h1.theme:varchar(50),h1.copies:int,pos:int) ->
{ answer(1,'White Christmas',50),       answer(2,'In the Summertime',31),
  answer(3,'Silent Night',30),          answer(4,'My Heart will Go On',25),
  answer(4,'Rock Around the Clock',25), answer(5,'I Will Always Love You',20),
  answer(5,'It\'s Now or Never',20),    answer(5,'We Are the World',20),
  answer(6,'If I Didn\'t Care',19) }
Info: 9 tuples computed.
```

[11] https://www.countryliving.com/life/news/a45720/white-christmas-song-history.

4.3. Prime Numbers

Problem Statement: Compute the first n prime numbers, where n is a natural bound.
A Solution: The sieve of Eratosthenes can be expressed in SQL by generating numbers and getting rid of those which can be divided by a lesser number. So, a natural number generator can be defined as a CTE, and then the factors for each number as another CTE. For a number to be a factor of another number it suffices to check that the division remainder is 0. The CTE `factor` below delivers pairs of x,f, where x is a number and f is one of its factors. The outcome selects each n for which there is only one factor.

```
/set bound 100
WITH
  number(n) AS
    (SELECT 0 UNION SELECT n+1 FROM number WHERE n < $bound$),
  factor(x, f) AS
    (SELECT x.n, f.n FROM number x, number f
     WHERE x.n > f.n AND f.n > 0 AND x.n mod f.n = 0)
SELECT x FROM (SELECT x, COUNT(*) c FROM factor GROUP BY x) WHERE c=1;
```

Note that DES includes user variables which can be used as parameters, such as `bound` in this example, which are set with the command `/set Variable Value`.
Datalog Formulation: Again, the translation seems to be more compact than the source SQL statement.

```
     number(0) /\ ( number(B) :- number(C), C<100, B=C+1 )
/\ ( factor(D,E) :- number(D), number(E), D>E, E>0, 0=D mod E )
=> group_by(factor(A,_F), [A], (_G=count,_G=1)).
```

Outcome:

```
answer(factor.x:int) ->
{ answer(2), answer(3), answer(5), answer(7), ..., answer(97) }
Info: 25 tuples computed.
```

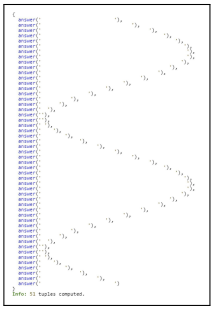
4.4. Graph of an Explicit Function

Problem Statement: Plot the sine function as depicted in the figure below.
A Solution: Here we define two CTE's: One working on the real plane (`plot`) which relates x and y coordinates via a function (parametrically specified with the user variable `f`), with square dimensions. The second CTE works on the "pixel" plane (`bars`) by building a string of spaces with as many spaces as pixels in the ordinates (horizontally represented instead of vertically), which will be truncated to the required length with respect to the value of the function at each x coordinate. Observe that strings in the display are quoted and ended with the characters "`'),`", so that it visually resembles the plot.

Both `plot` and `bars` are built in a similar way natural numbers are defined: a base case with the first point in the plane, and with a single space, respectively. The recursive case of `plot` adds pairs `(x+dx, fx)` for x coordinates up to its bound, whilst for `bars`, a space is added up to reaching its top length.

Once both relations are defined, the outcome query truncates (with the function `substr`: substring) the longest bar up to the value of the corresponding y coordinate for each x coordinate found in `plot`.

```
-- Dimension of the plot (d+1) rows and (d+1) columns:
/set d 50
-- Start x:
/set x0 0.0
-- End x:
/set x1 4*pi
-- Delta x:
/set dx ($x1$-$x0$)/$d$
-- Function:
/set f sin(xi)
WITH
```



```
  plot(x, fx) AS
   (SELECT $x0$ xi, $f$
    UNION
    SELECT x+$dx$ xi, $f$ FROM plot WHERE x <= $x1$),
  bars(bar) AS
   (SELECT ' '
    UNION
    SELECT bar||' ' FROM bars WHERE length(bar)<$d$)
SELECT substr((select max(bar) bar from bars), 1,
     round($d$*(fx-(SELECT min(fx) FROM plot))*(SELECT 1/(max(fx)-min(fx)) FROM plot)))
FROM plot;
```

Datalog Formulation: The following equivalent Datalog query includes the aggregate predicates `max/3` and `min/3`, which are syntax simplifications for a `group_by` predicate (e.g., `max(R,X,M)` $\equiv$ `group_by(R,[],M=max(X))`). Functions `concat(A,B)` (concatenation of `A` and `B` and written as `A||B` in standard SQL), `length(A)` (length of `A`), and `substr(A,P,N)` (substring of `N` characters of `A` taken from position `P`) are used.

```
   ( plot($x0$,F) :- F=sin($x0$) )
/\ ( plot(B,C) :- plot(D,_E), D=<$x1$, B=D+$dx$, C=sin(D+$dx$) )
/\ bars(' ') /\ ( bars(H) :- bars(G), length(G)<$d$, H=concat(G,' ') )
=> plot(_A,_B),
   max(bars(_K),_K,_H),
   min(plot(_L,_M),_M,_C),
   group_by(plot(_N,_O),[],(_F=1/(max(_O)-min(_O)))),
   A=substr(_H,1,round($d$*(_B-_C)*_F)).
```

4.5. Base Conversion

Problem Statement: Given the table `conversion(c:string, fb:int, tb:int)`, with natural numbers to be converted (`c`) from a base (`fb`) to another base (`tb`), write an SQL query returning a row for each element in the table `conversion`, with schema `(number_fb, fb, number_tb, tb)`, such that `number_fb` is the number in base `fb` equivalent to the number `number_tb` in base `tb`.

A Solution: This problem can be subdivided as usually in two subproblems: Convert to base 10, and then to the target base. We can assume a table `digits(n INT, d STRING)` containing tuples as $\{$ `(0,'0')`, `(1,'1')`, ..., `(10,'A')`, ...$\}$.

For converting to base 10, the approach is to recursively generate tuples with the inspected position i (index of the digit from right to left starting with 0) with a value corresponding to the number up to its i-th position. So, we can think of a schema for this first CTE as `to_base_10(c, fb, tb, s, i, n)`, where original parameters in `conversion` (`c`, `fb` and `tb`) are kept with their same names, `s` is the number truncated up to the $i-1$-th index, `i` is the index i, and `n` is the value (base 10) for the number `c` up to the i-th position. The original number is required in this schema because there will be in general several numbers to be converted, so that this first column will discriminate the number for which the conversion is made. A similar requirement applies to the original base `fb` and the target base `fb`.

For the sake of keeping neat the formulation, the base case for `to_base_10` starts at index -1, and the original number is appended (with the `||` operator) with a void -1-th position, delivering a value 0 for it. Otherwise, we could start at index 0, but calculations in the recursive case should be also included in the base case. This recursive case builds tuples from `to_base_10(c, fb, tb, s, i, n)` itself as follows: its first 3 columns are the original ones, as found in `to_base_10`. The number `s` (recall: a string) is right trimmed in 1 character; the index `i` is incremented, and its quantity (weighted with its position: `fb^(i+1)`) is added to `n`, where the value of the digit is taken from the `digits` table.

Converting a number in base 10 to another base is a somewhat similar problem, but with schema `to_base_b(c, fb, tb, n, st)`: the three original columns (`c`, `fb` and `tb`), the number in base 10 (`n`) and the most significant digits already processed (`st`). Here, the base case comes from the results for the relation `to_base_10`, selecting the value `n` for the final calculation (the length of the truncated number is 1: the last, most-significant character in the string). In this case, the index is not used because another approach is taken: dividing the current number by the target base and delivering the digit

corresponding to the remainder of the division as the next digit, from the less to the most significant digits. Then, the recursive case builds the outcome by appending the calculated digit to the left of the former outcome.

```
WITH
 to_base_10(c, fb, tb, s, i, n) AS (
   SELECT c, fb, tb, c || ' ', -1, 0
   FROM conversion
   UNION
   SELECT c, fb, tb, SUBSTR(s,1,LENGTH(s)-1), i+1,
          integer(n+(SELECT n
                     FROM digits
                     WHERE d=SUBSTR(to_base_10.s,LENGTH(to_base_10.s)-1,1) )*fb^(i+1))
   FROM to_base_10 WHERE LENGTH(s)>1
   ),
 to_base_b(c, fb, tb, n, st) AS (
   SELECT c, fb, tb, n, ''
   FROM to_base_10
   WHERE LENGTH(s)=1
   UNION
   SELECT c, fb, tb, n DIV tb,
          (SELECT d FROM digits WHERE n=to_base_b.n MOD to_base_b.tb) || st
   FROM to_base_b
   WHERE n MOD tb > 0
   )
SELECT c AS number_fb, fb AS fb, st AS number_tb, tb AS tb FROM to_base_b WHERE n=0;
```

number_fb	fb	number_tb	tb
FF	16	3333	4
1111	2	15	10
77	8	111111	2
FF	16	11111111	2

Datalog Formulation: The Datalog translation is more elaborated than in previous examples (but still, neater than its counterpart SQL formulation).

```
   ( to_base_10(E,F,G,H,-1,0) :- conversion(E,F,G), H=concat(E,' ') )
/\ ( to_base_10(I,J,K,L,M,N) :-
       to_base_10(I,J,K,O,P,Q), R=length(O), R>1, T=R-1, L=substr(O,1,T), M=P+1,
       digits(U,V), X=length(O)-1, V=substr(O,X,1), N=integer(Q+U*J**(P+1)))
/\ ( to_base_b(Y,Z,AA,AB,'') :- to_base_10(Y,Z,AA,AC,_AD,AB), length(AC)=1 )
/\ ( to_base_b(AE,AF,AG,AH,AI) :-
       to_base_b(AE,AF,AG,AJ,AK), AJ mod AG>0, AH=AJ div AG,
       digits(AM,AN), AM=AJ mod AG, AI=concat(AN,AK) )
=> to_base_b(A,B,D,0,C).
```

4.6. Universal Turing Machine

Problem Statement: Implement a universal Turing machine as a state transition system, following https://aturingmachine.com/. Possible transitions are specified in the input table `transitions(state, symbol, newstate, newsymbol, move)`, and the initial state is given in `initial_state(state, tape, pos)`. Here, `state` is a numeric value identifying the state, `tape` is a string containing a sequence of symbols `0`'s and `1`'s, `pos` is the position of the scanned symbol (the leftmost character of the tape is always the position 1 even when the tape can can grow indefinitely to the right or to the left), `move` can be either be `'L'` (to the left) or `'R'` (to the right).
Input Data: The two input tables are filled as shown next to implement a binary counting machine. It counts from `0` upwards without stopping. Inspecting its outcome can be done by adding a stop condition on the number of the transitions (steps).

```
INSERT INTO initial_state VALUES(0,' ',1);

-- (0,B) -> (1,B) Left: When a blank (B) is found we change to state 1:
INSERT INTO transitions VALUES(0,' ',1,' ','L');
-- (0,0) -> (0,0) Right: This state moves the tape to the right most digit:
INSERT INTO transitions VALUES(0,'0',0,'0','R');
-- The following transition is not needed:
-- (0,1) -> (0,1) Right: This state moves the tape to the right most digit
-- (1,B) -> (0,1) Right: If we change a Blank to a 1 we change back to state 0:
INSERT INTO transitions VALUES(1,' ',0,'1','R');
-- (1,0) -> (0,1) Right: If we change a 0 to a 1 we change back to state 0:
```

```
INSERT INTO transitions VALUES(1,'0',0,'1','R');
-- (1,1) -> (1,0) Left: If we change a 1 to a 0 we keep looking to the left:
INSERT INTO transitions VALUES(1,'1',1,'0','L');
```

A Solution: Implementing a universal Turing machine can be done with a recursive query where the base case contains the initial state, and the recursive case selects the next state in terms of the symbol in the current tape position. This must consider that the tape can grow both the left and to the right. For the last case, the string tape can be concatenated with a new symbol and the index at the first position does not change. For the former case, a new blank symbol is prepended and the index 1 is for this position.

```
WITH m(state,tape,pos,step) AS (
  SELECT state,tape,pos,1
  FROM initial_state
  UNION ALL
  SELECT
     -- State
     t.newstate,
     -- Tape
     CASE
       WHEN t.move='L' AND pos>1 THEN
         SUBSTR(tape,1,pos-1)||t.newsymbol||SUBSTR(tape,pos+1)
       WHEN t.move='L' AND pos=1 THEN
         ' '||t.newsymbol||SUBSTR(tape,pos+1)
       WHEN t.move='R' AND LENGTH(tape)=pos THEN
         SUBSTR(tape,1,pos-1)||t.newsymbol||' '
       WHEN t.move='R' AND LENGTH(tape)>pos THEN
         SUBSTR(tape,1,pos-1)||t.newsymbol||SUBSTR(tape,pos+1)
     END,
     -- Move
     CASE
       WHEN t.move='L' AND pos>1 THEN pos-1
       WHEN t.move='L' AND pos=1 THEN 1
       WHEN t.move='R'           THEN pos+1
     END,
     -- Step
     step+1
  FROM transitions t, m
  WHERE m.state=t.state AND
        SUBSTR(m.tape,m.pos,1)=t.symbol
        -- AND step < $max_step$  -- Use this as a stop condition
    )
SELECT step,state,pos,tape FROM m;
```

Datalog Formulation:

```
   (m(E,F,G,1) :-
      initial_state(E,F,G))
/\ (m(H,I,J,K) :-
      transitions(L,M,H,N,O), m(L,P,Q,R), '$substr'(P,Q,1,M), R<40, S=Q-1,
      '$substr'(P,1,S,T), '$concat'(T,N,U), V=Q+1, '$substr'(P,V,W), '$concat'(U,W,X),
      '$concat'(' ',N,Y), Z=Q+1, '$substr'(P,Z,AA), '$concat'(Y,AA,AB), AC=Q-1,
      '$substr'(P,1,AC,AD), '$concat'(AD,N,AE), '$concat'(AE,' ',AF), AG=Q-1,
      '$substr'(P,1,AG,AH), '$concat'(AH,N,AI), AJ=Q+1, '$substr'(P,AJ,AK),
      '$concat'(AI,AK,AL),
      '$case'([((O='L',Q>1),Q-1),((O='L',Q=1),1),(O='R',Q+1)],null,J), K=R+1,
      '$case'([((O='L',Q>1),X),((O='L',Q=1),AB),((O='R','$length'(P,AM),AM=Q),AF),
               ((O='R','$length'(P,AN),AN>Q),AL)],null,I))
=>  m(B,D,C,A).
```

5. Conclusions and Future Work

SQL puzzles for challenging students' skills have been presented and solved in the DES system, which has been selected because it helps students with syntax and semantic warnings about SQL queries. Since the system is based on intuitionistic logic programming, it allows for neater and simpler SQL

formulations to solve those puzzles because SQL queries are translated into a Hypothetical Datalog program. Both the syntax and the inference system for the Hypothetical Datalog system in [9] have been extended to support expressions in conditions and other meta-predicates needed for expressing SQL more complex queries. The implementation can be greatly enhanced in several ways; for example, replacing its prototype SQL parser with a more performant one, and supporting the semi-naïve differential optimization [19], which hugely improves performance for recursive problems, and fixing some known issues. Additionally, more puzzles will be developed for the enjoyment of students (and the teacher).

References

[1] M. Cook, Universality in elementary cellular automata, Complex Systems 15 (2004).

[2] A. Levitin, M.-A. Papalaskari, Using puzzles in teaching algorithms, SIGCSE Bull. 34 (2002) 292–296.

[3] J. Celko, Joe Celko's SQL Puzzles and Answers, Second Edition (The Morgan Kaufmann Series in Data Management Systems), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[4] S. Wikipedia, Esoteric Programming Languages: Brainfuck, Intercal, Befunge, Esoteric Programming Language, Kvikkalkul, One Instruction Set Computer, Unlambda, Malbo, University-Press Org, 2013. URL: https://books.google.es/books?id=4X_hngEACAAJ.

[5] F. Sáenz-Pérez, DES: A Deductive Database System, Electronic Notes on Theoretical Computer Science 271 (2011) 63–78.

[6] F. Sáenz-Pérez, Applying constraint logic programming to sql semantic analysis, Theory and Practice of Logic Programming 19 (2019) 808–825. doi:10.1017/S1471068419000206.

[7] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez, Declarative Debugging of Wrong and Missing Answers for SQL Views, in: Eleventh International Symposium on Functional and Logic Programming (FLOPS 2012), LNCS 7294, Springer, 2012, pp. 73–87.

[8] A. J. Bonner, Hypothetical Datalog: Complexity and Expressibility, Theoretical Computer Science 76 (1990) 3–51.

[9] F. Sáenz-Pérez, Intuitionistic Logic Programming for SQL, in: M. V. Hermenegildo, P. Lopez-Garcia (Eds.), Logic-Based Program Synthesis and Transformation, Springer International Publishing, Cham, 2017, pp. 293–308.

[10] F. Arni, K. Ong, S. Tsur, H. Wang, C. Zaniolo, The Deductive Database System LDL++, TPLP 3 (2003) 61–94.

[11] F. Sáenz-Pérez, Tabling with support for relational features in a deductive database, ECEASST 55 (2012).

[12] A. J. Bonner, L. T. McCarty, Adding Negation-as-Failure to Intuitionistic Logic Programming, in: E. L. Lusk, R. A. Overbeek (Eds.), Proceedings of the North American Conference on Logic Programming (NACLP), The MIT Press, 1990, pp. 681–703.

[13] F. Sáenz-Pérez, Restricted predicates for hypothetical datalog, Electronic Proceedings in Theoretical Computer Science 200 (2015) 64–79.

[14] F. Sáenz-Pérez, Implementing tabled hypothetical datalog, in: Proceedings of the 25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI'13, 2013, pp. 596–601.

[15] J. D. Ullman, Principles of Database and Knowledge-Base Systems, Volume II, Computer Science Press, 1988.

[16] ISO/IEC, SQL:2016 ISO/IEC 9075-1:2016 Standard, 2016.

[17] L. Sterling, E. Shapiro, The Art of Prolog (2Nd Ed.): Advanced Programming Techniques, MIT Press, Cambridge, MA, USA, 1994.

[18] S. Nieva, F. Sáenz-Pérez, J. Sánchez, HR-SQL: An SQL Database System with Extended Recursion and Hypothetical Reasoning. *Ongoing Work*, in: XV Jornadas sobre Programación y Lenguajes, PROLE2015 (SISTEDES), 2015, pp. 1–15.

[19] I. Balbin, K. Ramamohanarao, A generalization of the differential approach to recursive query evaluation, The Journal of Logic Programming 4 (1987) 259 – 262.