From relational to deductive databases

Fernando Sáenz-Pérez*

Grupo de programación declarativa (GPD),
Dept. Ingeniería del Software e Inteligencia Artificial,
Universidad Complutense de Madrid, Spain
fernan@sip.ucm.es

Abstract. This paper highlights the limits of current implementations of SQL and shows how they can be overcome with deductive approaches. Deductive query languages subsuming relational ones must include constructors to allow users for submitting queries semantically equivalent to their relational counterparts. As deductive languages are based on first order predicate logic, they are able to capture relational semantics, which involves negation. In addition to negation, as e.g. needed to express set difference, usual relational outer join statements and aggregate functions must be taken into account for such subsuming deductive languages. We base our presentation on the grounds of DES (Datalog Educational System), a deductive database system that integrates both deductive and relational database languages in a common inference engine and system.

Keywords: Relational databases, Deductive databases, SQL, Datalog, DES

1 Introduction

Contributions from mathematics to data management systems have revealed important landmarks as exemplified by the work in the relational database model by Codd [11]. By that time, logic programming (started with the work of John McCarthy) lead to the Prolog language with well-known efforts from Edinburgh and Marseille. This, in turn, branched in multiple disciplines including the work on deductive database systems, pioneered by the Datalog [1] query language. That decade was fruitful for data management in that also the SQL language was firstly implemented in IBM's System R.

Relational data model and its extensions have been of great success as they are the base of current, widely-used, relational database management systems and its ubiquitous SQL query language. But, despite its acknowledged benefits, SQL is also claimed as an error-prone and unclear language [12]. Instead, Datalog has been claimed as a pure declarative query language [1, 10]; i.e., it is able to allow a user to specify *what* he wants rather than *how* it must be performed.

* This author has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502)

Nonetheless, SQL builds on many features which as a whole are not usual in deductive databases, making it more practical.

With the advent of new interest on deductive databases and emerging companies promoting deductive technologies, this might change in a future. Datalog has been extensively studied and is gaining a renowned interest thanks to their application to ontologies [9], semantic web [13], social networks [17], policy languages [5], and even for optimization [15]. In addition, current companies as LogicBlox, Exeura, Semmle, and Lixto embody Datalog-based deductive database technologies in the solutions they develop. All these facts reveal the actual resurgence of Datalog in both academia and industry.

Thus, the aim of this paper is to try to look at deductive databases from a renowned viewpoint as more features are being added making them more appealing. Section 2 introduces the concrete deductive system we base our presentation on, which allows confronting both Datalog and SQL query languages, a subject covered in the core Section 3 of this paper. The paper ends by drawing some conclusions in Section 4.

2 Datalog Educational System

The Datalog Educational System (DES) [19] is a free, open-source, multiplatform, portable, in-memory, Prolog-based implementation of a deductive database system. DES 2.3 [18] is the next shortcoming release (expected by May), which enjoys Datalog and SQL query languages, full recursive evaluation with tabling, full-fledged arithmetic, stratified negation [20], ODBC connections and novel approaches to Datalog and SQL declarative debugging [6, 8], test case generation for SQL views [7], null values support, (tabled) outer join and aggregate predicates and functions [18]. However, as the system was intended for educational purposes, it is not targeted at performance and also lacks features such as concurrency, security and others that a practical database system must enjoy.

2.1 Datalog

The Datalog version considered in DES is as follows:

- A Datalog *program* consists of a set of rules. A program may contain line remarks, which start with the symbol %.
- A *rule* has the form `head :- body`, or simply `head`, ending with a dot.
- A *head* is a positive atom including no built-in predicate symbols.
- A *body* contains conjunctions (denoted by "`,`") as well as disjunctions (denoted by "`;`") of literals (with usual associativity and priority).
- A *literal* is either an atom, or a negated atom (`not(`*`Atom`*`)`), or a built-in (literals are also referred to as goals). *Basic* literals only include atoms, negated atoms, and outer join built-ins.
- An *atom* is an atomic formula [2] restricted to have variables or constants as arguments.

– A *variable* is a program symbol starting with either an uppercase letter or an underline.
– A *constant* is a program symbol either starting with a lowercase letter or being a sequence of characters delimited by single quotes.
– A *query* is a literal (some built-ins are exceptions, as will be shown in Section 3.4, and include other atoms as arguments). In addition, temporary views can also be submitted as queries, as will be introduced in Subsection 2.3.

Compound terms are not allowed but as arithmetic expressions, which can occur in certain built-ins (for writing arithmetic expressions and conditions).

Datalog programs are typically consulted from files, and queries are typed at the system prompt. The answer to a query is the set of facts matching the query which are deduced in the context of the database. A query with different variables for all its arguments gives the whole set of facts (meaning) defining the queried relation. If a query contains a constant in an argument position, it means that the query processing will select the facts from the meaning of the relation such that the argument position matches the constant (i.e., analogous to a *select* relational operation with an equality condition). If a given variable occurs more than once in a query, the corresponding predicate arguments are required to match.

DES implements Datalog with stratified negation as described in [20] with safety checks [20, 22]. Stratified negation broadly means that negation is not involved in a recursive computation path, although it can use recursive rules. The system can compute a query Q in the context of a program that is restricted to the dependency graph (which shows the computation dependencies among predicates) built for Q so that a stratification can be found. This means that, even when a program could be actually non-stratifiable, a query involving a subset of the program might be safely computable, provided that a suitable stratification can be found for its dependency subgraph.

Evaluation of queries is ensured to be terminating as long as no infinite predicates/operators are considered. Currently, only the infix operator `is` represents an infinite relation and can deliver unlimited pairs (other built-ins, as comparison operators, demand their arguments to be ground). For instance, let's consider the rules `p(0).` and `p(X) :- p(Y), X is Y+1.` Then, the query `p(X)` is not terminating since its meaning is infinite ({`p(0), p(1), ...`}).

2.2 SQL

DES covers a reasonable set of SQL following ISO standard SQL:1999 (further revisions of the standard cope with issues such as XML, triggers, and cursors, which are outside of the scope of DES). There is provision for the DDL (data definition language – `CREATE TABLE`, ...), DML (data manipulation language – `INSERT INTO`, ...) and DQL (data query language – `SELECT`, ...) parts of the language[1].

[1] Note that we distinguish, as opposed to common use, DQL and DML. Usually, DQL, as we understand it, is rather included in DML.

Some supported features include:

- Arithmetic expressions.
- Null values.
- Outer joins (left, right and outer, no limitations on applications).
- Aggregates.
- Duplicates and duplicate elimination.
- Correlated queries.
- Recursive queries.
- Integrity constraints (primary keys and referential integrity).
- Types (domains).

Also, there are some important differences w.r.t. SQL:

- User identifiers are case sensitive (as table and column names).
- As the underlying engine works on a two-valued logic, logical expressions over nulls do not behave exactly as defined in the SQL standard.
- Duplicates are disabled by default (but can be enabled as well)

Limitations can be found in [18], and to name a few, the following are unsupported, up to now:

- `ALL`, `SOME` and `ANY`.
- Query result coercions to values in projection list.
- String expressions.
- `ORDER BY` clause.

SQL DQL statements are compiled to and executed as Datalog programs (basics can be found in [20]), and relational metadata for DDL statements are kept. Submitting a DQL query amounts to 1) parse it, 2) compile to a Datalog program including the relation `answer`$/n$ with as many arguments as expected from the SQL statement, 3) assert this program, and 4) submit the Datalog query `answer`$(X_1, \ldots, X_n)$, where $X_i : i \in \{1, \ldots, n\}$ are n fresh variables. After its execution, this Datalog program is removed. On the contrary, if a DDL statement defining a view is submitted, its translated program and metadata do persist.

2.3 System Features

As an interactive (text-based) system, DES accepts user inputs as both Datalog and SQL queries, and commands. Commands are isolated from the database signature, i.e., name clash is avoided even when a relation gets the same name than a command because commands are preceded with the symbol "/". There is provision for rule database commands (inserting, deleting and listing both programs and single rules), operating system commands (related to the OS file system and OS commands and programs), extension table commands (tabling

information), log commands (logging system output to files), informative commands (predicate dependency graph, stratification, system status, help and others), and miscellanea commands (halting the system, quitting the session and invoking Prolog).

There are available some usual built-in comparison operators (`=`, `\=`, $>$, ...). When being solved, all these operators demand ground (variable-free) arguments (i.e., no constraints are allowed up to now) but equality, which performs unification. In addition, arithmetic expressions are allowed via the infix operator `is`, which relates a variable with an evaluable arithmetic expression. The result of evaluating this expression is assigned/compared to the variable. The predicate `not/1` implements stratified negation. Other built-ins include outer joins and aggregates.

As a user facility, *temporary views* are provided: a novel feature that allows to write compound queries on the fly (as, e.g., conjunctions and disjunctions). A temporary view is a rule which is added to the database, and its head is submitted as a query and executed. Afterwards, the rule is removed. For instance, given the relations `a`/1 and `b`/1 defined by facts, the view `d(X) :- a(X), not(b(X))` computes the set difference between the instance relations `a` and `b`.

In addition, automatic temporary views, *autoviews* for short, are temporary views which do not need a head. When submitting a compound query (any allowed body), a new temporary relation, named `answer` (a common name for the relation result [1], Paradox, ...), is built with as many arguments as relevant variables occur in the conjunctive query. `answer` is a reserved word and cannot be used for defining other relations. The conjunctive query `a(X), b(Y)` is an example of an autoview, which computes the Cartesian product of `a` and `b`.

Both Datalog and SQL languages are provided sharing the same database, as SQL statements are compiled to Datalog rules and solved by the deductive inference engine. Moreover, Datalog programs can seamlessly refer to objects created in the SQL side, as tables and views. Whereas RDBMS's keep the *extensional* database (EDB, defined by tables) isolated from the *intensional* database (IDB, defined by views), i.e., their signatures are disjoint, in a deductive database both parts are joined since a relation (predicate) is defined by rules. In addition to access common data defined by both SQL and Datalog from within DES, another possibility is to interact with external databases, as shown in next subsection, since ODBC connections are available.

2.4 ODBC Connections

DES provides support for connections to RDBMS's in order to provide data sources for relations. This means that a relation defined in a RDBMS as a view or table is allowed as any other relation defined via a predicate in the deductive database. Then, computing a query can involve computations both in the deductive inference engine and in the external RDBMS SQL engine. Such relations become first-class citizens in the deductive database and, therefore, can be queried in Datalog. These queries, in turn, can refer to both predicates, tables and views in a transparent way. If the relation is a view, it will be processed

by the SQL engine. When an ODBC connection is opened, all SQL statements are redirected to such connection, so DES does not longer process such statements. This also means that all the SQL features of the connected RDBMS are available.

However, as a caveat, notice that data from SQL queries are cached in an extension table during Datalog computations, and such data are not retrieved anymore until this cache is cleared (either explicitly with the command `/clear_et` or because a command or statement invalidate its contents, as a SQL update query). Therefore, it could be possible to access outdated data from a Datalog query should the external database is modified.

In addition, it is not recommended to mix Datalog and SQL data. Current release allows to assert rules for predicates with the same name and arity as existing RDBMS's tables and/or views. So, although on the DES side, rules are known, they are not on the RDBMS side and thus a SQL statement will answer data in the external RDBMS, without caring for possible Datalog rules intersecting those. This is in contrast to the plain DES management of data (in the absence of an ODBC connection): the DES engine is aware of any changes to data, both from Datalog and SQL sides.

2.5 Query Compilation

As a final remark for this section, since SQL statements are compiled to Datalog, it is an interesting exercise to inspect how the DES compiler translates SQL to Datalog programs. This can be seen by enabling compilation listings at the system prompt by issuing the command `/show_compilations on`. Other possible way is consulting a view schema (command `/dbschema` *`view_name`*): the view text is listed together with its compilation to Datalog rules (without an opened ODBC connection).

3 Comparing SQL and Datalog

Here, we compare Datalog and SQL as query languages hopefully highlighting the more compact and neat formulations and better expressiveness that Datalog enjoys. In what follows, all statements (SQL and Datalog) can actually be submitted in DES at its command prompt, sharing predicates, tables and views.

3.1 Defining Relations

First task to show is declaring metadata information about tables, that is, relation variables (schema) as opposed to relation values (instances) (following nomenclature in [12]). Whilst in SQL we use the `CREATE TABLE` statement for such purpose, in DES we use type declarations instead. So, the following two inputs produce equivalent results:

```
CREATE TABLE s(sno INT, name VARCHAR(10));
:-type(s(sno:int, name:varchar(10))).
```

Type declaration in DES predicates is optional and different from other systems so that is more geared towards compatibility with SQL. Such declarations should be understood as program assertions (as in Prolog) rather than becoming part of the language (assertions are not allowed in rule bodies). Another assertion allows to defining a primary key with `:-pk(s,[sno])`.

One can also define a propositional relation (0-arity predicate) with `:-type(p)`. And as a distinguishing feature, such propositional relations can be queried from both SQL and Datalog (in the former case, we depart from the relational model by adding this feature).

3.2 Populating Relations

From the SQL viewpoint, populating relations are usually performed with `INSERT INTO` statements. Datalog facts, defining the extensional part, are typically and simply written in a text file and then consulted with a command, as in `/consult suppliers`, where `suppliers.dl` is the intended file to be consulted (where extension `.dl` is optionally specified in the command). Also, the command `/assert` *`Rule`* is provided for inserting a rule. The following are equivalent:

```
insert into employee values('Smith','Sales',1500);
/assert employee('Smith','Sales',1500).
```

`INSERT` statements are considered static in the sense they are not mixed with query solving; but to a point: the statement can contain a query which acts as a data source from tables and views. To enable termination in such queries, if a target table `t` also occurs as a source relation, then source data for `t` are the ones before insertions by the inserting statement. So, the second statement below is terminating:

```
insert into s values(1,'1st Supplier');
insert into s select sno+1,'2nd Supplier' from s;
```

This kind of statements are processed in DES as follows: First, compile the `SELECT` query to Datalog and solve this query. Then, its results are added to the target table.

Such programmatic populations are not allowed in Datalog, but they are rather subject of further work. Insertions and deletions could be allowed, respectively, by prepending '+' and '-' to base relations, as in LDL++ [3].

3.3 Basic Queries

Basic extended relational algebra operations can be applied as follows, using both Datalog and SQL. Note that in Datalog, an explicit denotation of relation arguments is required. We assume that duplicated are disabled, so that outputs are sets as in the relational model (duplicates can be enabled on user demand).

- $\pi_{name}(s)$: Projecting the second argument of s.

```
projection(Y) :- s(X,Y).
WITH projection(name) AS (SELECT name FROM s)
  SELECT * FROM projection;
```

A renaming operation is also performed in the first case to get the relation name `projection`. Note that such rule is known in DES as a temporary view and can be submitted from the prompt as such. In SQL, an analogous operation is performed via a `WITH` statement (parentheses have been added to aid legibility; though, they are not needed). For the sake of brevity, however, we'll not apply renamings from now on. Also, as only single-line command-prompt inputs are allowed, ending dots and semicolons are optional and we can omit them safely.

- $\sigma_{sno=1}(s)$: Selecting the `name` value from `s` such that `sno` is 1

```
s(1,Y)
SELECT name FROM s WHERE sno=1
```

- $s \times sp$: Cartesian product of relations `s` and `sp`

```
s(X,Y), sp(U,V)
SELECT * FROM s, sp
```

- $s \bowtie sp$: Natural inner join of relations s and sp

```
inner_join(X,Y,V) :- s(X,Y), sp(X,V)
SELECT * FROM s NATURAL INNER JOIN sp
```

In the Datalog case, it is needed to use the temporary view to select the arguments corresponding to the natural join.

- $s \cup q$: Set union of relations `s` and `q`

```
s(X,Y) ; q(X,Y)
SELECT * FROM s UNION SELECT * FROM q;
```

Note that a view is evaluated in the context of the database; so, if there are more rules already defined with the same name and arity of the rule head, the evaluation of the view will return its answer considering the database already loaded. For instance, the temporary view `s(X,Y) :- q(X,Y)` is equivalent to the former statements as it computes the union of `s` and `q`. This rule is a tuple source from the data in `q`, which will form part of the answer together with the tuples defined by the already defined facts of `s`. This is no longer true in the relational case because of the isolation of the extensional and intensional parts of the database (a view cannot be created with the same name as a table).

- $s \backslash q$: Set difference of relations `s` and `q`

```
s(X,Y), not(q(X,Y))
SELECT * FROM s EXCEPT SELECT * FROM q
```

What follows from these pretty small examples is that it can be argued that SQL syntax was chosen to better follow a natural speech, whereas Datalog is more related to mathematical notation and therefore more concise in some cases, a point further emphasized in next subsections.

3.4 Outer Joins

DES provides the three outer join operations: left, right and full outer joins. The left (resp. right, and full) outer join corresponds to the built-in `lj(A,B,C)` (resp. `rj(A,B,C)`, and `fj(A,B,C)`), with `A`, `B`, basic literals, and `C` a literal. Built-in `lj(A,B,C)` computes the cross-product of tuples in the meaning of `A` and `B` that satisfy literal `C`, extended with those tuples in the meaning of `A` for which `C` is not true, so that they include nulls for `B`'s arguments. So, next inputs are equivalent:

```
lj(s(X,Y),sp(U,V),X=U)
SELECT * FROM s LEFT JOIN sp ON s.sno=sp.sno
```

Compared to current RDBMS implementations, there are no restrictions at all on what form the condition (here, literal `C`) can take.

A *join* condition has not to be missed with a *where* condition. The above query `lj` query is not equivalent to `lj(s(X,Y),sp(X,V),true)` (Notice that the variable `X` is shared for relations `s` and `sp`.) This query could be written in SQL as follows: `SELECT * FROM s LEFT JOIN sp WHERE s.sno=sp.sno`

But note that Datalog admits a more neat formulation if both conditions are needed, say[2]:

```
lj(s(X,Y),sp(U,Y),X=U)
SELECT * FROM s LEFT JOIN sp ON s.sno=sp.sno WHERE name=pno
```

Outer join relations can be nested as well, as: [3]:

```
lj(s(X,Y),rj(q(U,V),sp(Z,W),U=Z),X=U)
SELECT * FROM s LEFT JOIN (q RIGHT JOIN sp ON q.sno=sp.sno)
ON s.sno=q.sno
```

Further, some RDBMS's as DB2 does not allow the above and otherwise need a longer formulation. In Datalog, variables are scoped all way long, so that *where* conditions can refer to any nested relation (note the multiple occurrences of variable `Y`).

```
lj(s(X,Y),rj(q(U,Y),sp(Z,Y),U=Z),X=U)
SELECT * FROM s LEFT JOIN
(SELECT * FROM q RIGHT JOIN sp ON q.sno=sp.sno WHERE q.name=sp.pno)
ON s.sno=q.sno WHERE s.name=q.name
```

3.5 Queries with Aggregates

DES provides aggregate *functions* to be used in a similar way as in SQL. In addition, aggregate *predicates* and a `group_by` predicate are also provided. Noticeably, a *group by* operation can be constructed automatically without the need of specifying a `group_by` predicate, as will be shown below. Again, for the sake of comparison, SQL formulations also accompany to Datalog ones.

[2] Up to meaningfulness and type correctness.

[3] Incidentally, MS Access neither allows this combination of outer joins nor full joins.

Aggregate Functions An aggregate function can occur in expressions and returns a value, as in `R=1+sum(X)`, where `sum` is expected to compute the cumulative sum of possible values for `X`, and `X` has to be bound in the context of a `group_by` predicate (cf. next paragraph), wherein the expression also occurs.

Predicate `group_by` This predicate encloses a query for which a given list of variables builds answer sets (groups) for all possible values of these variables. If we consider the relation `employee(`*`Name`*`,`*`Department`*`,`*`Salary`*`)`, the number of employees for each department can be counted with the query:

```
group_by(employee(N,D,S), [D], R=count)
SELECT Department,COUNT(*) FROM employee GROUP BY Department
```

If employees are not yet assigned to a department (i.e., a null value in *`Department`*), then this query behaves as a SQL user would expect: excluding those employees from the count outcome. If we rather want to count active employees (those with assigned salaries), we can use the query:

```
group_by(employee(N,D,S), [D], R=count(S))
SELECT Department,COUNT(Salary) FROM employee GROUP BY Department
```

Conditions including aggregates on groups (cf. `HAVING` conditions in SQL) can be stated as well. E.g., for retrieving departments with more than one active employee:

```
group_by(employee(N,D,S), [D], count(S)>1)
SELECT Department FROM employee GROUP BY Department
HAVING COUNT(Salary)>1
```

Conditions including no aggregates on tuples (cf. `WHERE` conditions in SQL) of the input relation (cf. SQL `FROM` clause) can also be used. For instance, the following query computes the number of employees and the average salary by department, for salaries greater than 1,000:

```
group_by((employee(N,D,S),S>1000), [D], (C=count(S),A=avg(S)))
SELECT Department,COUNT(Salary),AVG(Salary) FROM employee
WHERE Salary>1000 GROUP BY Department
```

Observe that the following query is not equivalent to the last one, since variables in the input relation are not expected to be bound after a grouping computation, and it raises a run-time exception upon execution:

```
group_by(employee(N,D,S), [D], (C=count(S),A=avg(S))), S>1000
```

The following example shows predicate `group_by` really admits a more compact representation than its SQL counterpart:

```
group_by(employee(N,D,S),[D],(C=count(S);C=sum(S)))
SELECT Department,COUNT(Salary) FROM employee GROUP BY Department
  UNION
SELECT Department,SUM(Salary) FROM employee GROUP BY Department
```

Aggregate Predicates An aggregate predicate returns its result in its last argument position, as in `sum(P,X,R)`, which binds `R` to the cumulative sum of values for `X`, provided by the input relation `P` which in particular explicitly includes variable `X`. These aggregate predicates simply allow another way of expressing aggregates, in addition to the way explained just above. For instance, for counting active employees, the following query is possible:

```
count(employee(N,D,S),S,T)
SELECT COUNT(Salary) FROM employee
```

If we rather omit the second argument of `count`, this predicate behaves as `COUNT(*)` in SQL. (In this example, it would count *all* the employees, not only those with assigned salary.)

A `group_by` operation is simply specified by including the grouping variable(s) in the head of a clause, as in the following view, which computes the number of active employees by department:

```
v(D,C) :- count(employee(N,D,S),S,C)
SELECT Department,COUNT(Salary) FROM employee GROUP BY Department
```

Correlated aggregates are also allowed, including them as another goal of the first argument of the aggregate predicate as, e.g., in the following view, which computes the number of employees that earn more than the average company salary:

```
v(D,C) :- count((employee(N,D,S),avg(employee(N1,D1,S1),S1,A),S>A),C)
SELECT Department,COUNT(Salary) FROM employee
WHERE Salary > (SELECT AVG(Salary) FROM employee)
GROUP BY Department
```

Note that last Datalog query uses different variables in the same argument positions for the two occurrences of the relation `employee`. Compare this to the following query, which computes the number of employees so that each one of them earns more than the average salary of his corresponding department. Here, the same variable name `D` has been used to refer to the department for which the counting and average are computed:

```
v(D,C) :- count((employee(N,D,S),avg(employee(N1,D,S1),S1,A),S>A),C)
SELECT Department,COUNT(*) FROM employee e1
WHERE e1.Salary > (SELECT AVG(Salary) FROM employee e2
                   WHERE e1.Department=e2.Department)
GROUP BY Department
```

3.6 Recursive Queries

Let's consider a classical transitive closure problem: Given a graph defined by the relation `edge(Origin,Destination)`, find the minimum path between any pair of reachable nodes assuming that the length of an edge is 1. A possible recursive SQL formulation follows:

```
   CREATE OR REPLACE VIEW
     shortest_paths(Origin,Destination,Length) AS
   WITH RECURSIVE path(Origin,Destination,Length) AS
     (SELECT edge.*,1 FROM edge)
   UNION
     (SELECT path.Origin,edge.Destination,path.Length+1
      FROM path,edge
      WHERE path.Destination=edge.Origin AND
            path.Length < (SELECT COUNT(*) FROM edge) )
   SELECT Origin,Destination,MIN(Length)
   FROM path
   GROUP BY Origin,Destination;
-- Query:
   SELECT * FROM shortest_paths;
```

But this formulation is not allowed in several RDBMS implementations (e.g., DB2 and SQL Server) because of several reasons, either because `GROUP BY`, `HAVING`, duplicate elimination (as in `UNION`) or aggregates are not allowed in the recursive part of queries (DES, though, does). The very same problem can be formulated in Datalog as:

```
  path(X,Y,1) :- edge(X,Y).
  path(X,Y,L) :- path(X,Z,L0), edge(Z,Y),
                 count(edge(A,B),Max), L0<Max, L is L0+1.
% Query:
  shortest_paths(X,Y,L) :- min(path(X,Y,Z),Z,L).
```

Current RDBMS's following SQL:1999 require stratification w.r.t. negation and aggregates to support recursion. Negation in SQL occurs for `NOT EXIST` and `EXCEPT` clauses (note that conditions such as `NOT(A>B)` become `A<=B` and are not considered therefore as negation). But stratification means that several graph algorithms cannot be expressed in SQL [22]. Further, linear recursion in SQL restricts to one the number of allowed recursive calls. For instance, Fibonacci numbers cannot be computed.

Related also to linearity, isolating IDB and EDB in SQL also poses problems. For instance, the basic transitive closure shown next is not possible in current RDBMS's implementations as nonlinear recursion is involved (DES, however, does allow it).

```
WITH paths(Origin,Destination) AS
    (SELECT 1,2)
  UNION
    (SELECT 2,3)
  UNION
    (SELECT p1.Origin,p2.Destination
     FROM paths p1,paths p2
     WHERE p1.Destination = p2.Origin)
  SELECT Origin,Destination FROM paths;
```

An equivalent query in Datalog follows:

```
path(X,Y) :- (X=1,Y=2) ; (X=1,Y=3) ; (path(X,Z),path(Z,Y))
```

To end this subsection, let's recall that one of the successful outcomes of the Datalog community was the magic set transformation [4] which is used to implement recursion in RDBMS's (Starburst [16] was the first non-commercial RDBMS to implement this whereas IBM DB2 was the first commercial one).

4 Conclusions

Under the risk of falling into subjectiveness, one might argue that this paper has shown how SQL statements can be expressed with Datalog rules with more neat and compact formulations. It is commonly acknowledged that shorter codings improve readability and program maintenance, and examples do show the shorter codings of Datalog w.r.t. SQL. In addition, its more mathematical syntax allows more understandable programs. However, from another perspective, it could be criticized that shorter codings amounts to a higher semantics/syntax ratio, that is, a formula entails a heavier semantic load; but this is also rather a debate.

So, compared to the widely-used relational database language SQL, Datalog adds two main advantages. First, its clean semantics allows to better reason about problem specifications. Its more compact formulations, notably when using recursive predicates, allow better understanding and program maintenance. Second, it provides more expressivity because the linear recursion limitation in SQL is not imposed. In fact, multiple recursive calls can be found in a deductive rule body. Stratification also restricts expressiveness (think, for instance, of how to formulate a two-people game defined by the single rule `winning(X) :- move(X,Y), not(winning(Y))` with stratified semantics). Although we have used DES, a system implementing stratified negation, other engines relaxing this restriction could be connected, as those implementing stable models (on which DLV is founded) [14] and well-founded semantics (on which ASP is founded) [21]. In addition, we have seen that this system does not impose any limitations on SQL statements as current RDBMS's do (as long as they can be expressed in Datalog, too), so that users do not have to struggle about how to formulate a given query trying to overcome such limitations by contrived reformulations.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. Krzysztof R. Apt. Introduction to logic programming. Technical report, University of Texas at Austin, Austin, TX, USA, 1988.
3. Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. The Deductive Database System LDL++. *TPLP*, 3(1):61–94, 2003.
4. François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15. ACM, 1986.

5. Moritz Becker, Cedric Fournet, and Andrew Gordon. Design and Semantics of a Decentralized Authorization Language. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 3–15, Washington, DC, USA, 2007. IEEE Computer Society.
6. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. A Theoretical Framework for the Declarative Debugging of Datalog Programs. In *International Workshop on Semantics in Data and Knowledge Bases (SDKB)*, volume 4925 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2008.
7. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Applying Constraint Logic Programming to SQL Test Case Generation. In *Proc. International Symposium on Functional and Logic Programming (FLOPS'10)*, volume 6009 of *Lecture Notes in Computer Science*, 2010.
8. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Algorithmic Debugging of SQL Views. In *Ershov Informatics Conference (PSI'11)*, Lecture Notes in Computer Science. Springer, 2011. In Press.
9. Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog±: a unified approach to ontologies and integrity constraints. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 14–30, New York, NY, USA, 2009. ACM.
10. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Tran. on Knowledge and Data Engineering*, 1(1):146–166, 1989.
11. E.F. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM*, 13(6):377–390, June 1970.
12. C J Date. *SQL and relational theory: how to write accurate SQL code.* O'Reilly, Sebastopol, CA, 2009.
13. Richard Fikes, Patrick J. Hayes, and Ian Horrocks. OWL-QL - a language for deductive query answering on the Semantic Web. *J. Web Sem.*, 2(1):19–29, 2004.
14. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
15. Sergio Greco, Irina Trubitsyna, and Ester Zumpano. NP Datalog: A Logic Language for NP Search and Optimization Queries. *Database Engineering and Applications Symposium, International*, 0:344–353, 2005.
16. Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. *SIGMOD Rec.*, 23:103–114, May 1994.
17. Royi Ronen and Oded Shmueli. Evaluating very large Datalog queries on social networks. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 577–587, New York, NY, USA, 2009. ACM.
18. Fernando Sáenz-Pérez. Datalog Educational System V2.3, May 2011. `des.sourceforge.net/`.
19. Fernando Sáenz-Pérez. DES: A Deductive Database System. *Electronic Notes on Theoretical Computer Science*, 271:63–78, March 2011.
20. Jeffrey D. Ullman. *Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies).* Computer Science Press, 1988.
21. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.
22. C. Zaniolo, S. Ceri, C. Faloutsos, R T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.