Towards Bridging the Expressiveness Gap Between Relational and Deductive Databases

Fernando Sáenz-Pérez[1]*

Grupo de programación declarativa (GPD),
Dept. Ingeniería del Software e Inteligencia Artificial,
Universidad Complutense de Madrid, Spain[1]

Abstract: SQL technology has evolved during last years, and systems are being more powerful and scalable. However, there exist yet some expressiveness limitations that can be otherwise overcome with inputs from deductive databases. This paper focuses on both practical and theoretical expressiveness issues in current SQL implementations that are overcome in the Datalog Educational System (DES), a deductive system which also includes extended SQL queries with respect to the SQL standard and current DBMS's. Also, as external database access and interoperability are allowed in DES, results from the deductive field can be tested on current DBMS's. For instance: Less-limited SQL formulations as non-linear recursive queries, novel features as hypothetical queries, and other query languages as Datalog and Extended Relational Algebra. In addition, some notes on performance are taken.

Keywords: Relational Databases, Deductive Databases, SQL, Datalog, Expressiveness

1 Introduction

Deductive database systems extend ("relational") database management systems (DBMS's) by including a more powerful query language based on logic. Datalog (and its extensions as Datalog with negation, uninterpreted function symbols, disjunctive heads, constraints, . . . [RU93]) became the *de-facto* deductive query language. This language has been extensively studied and nowadays is regaining an increased interest in different database application areas [Got12].

Motivations for researching in deductive databases include clean semantics of logic-based approaches, neat formulations, and expressiveness gains. Provided that the deductive data model (function-free case) meets the relational one with respect to the data structures, it is possible to query the same database (either intensional or extensional, and either deductive or relational) with different languages, as Datalog, SQL, and Relational Algebra (RA).

Language expressiveness can be seen from two points of view: Expressive power (*theoretical* expressiveness), and concisely and readily (*practical* expressiveness). Nowadays, one can face several obstacles when formulating SQL queries because of a number of reasons, including, e.g.: parsing requirements and restricted syntax constructions for practical expressiveness, and limited

* 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)

set of operators and recursion limitations for theoretical expressiveness. Whereas the user can work around practical expressiveness obstacles, usually contrived formulations are reached. On the other hand, theoretical expressiveness even makes impossible to formulate a given query.

In this work, we present an ongoing work that enables more powerful and less restrictive forms of SQL queries (w.r.t. current DBMS's) in the deductive system DES (Datalog Educational System) [SCG11], a system targeted at teaching SQL, RA and Datalog at class rooms. It can be argued that less restrictive SQL queries enable students a more rapid learning curve because less problems arise in writing such queries. Also, we consider extended SQL, RA and Datalog languages with novel features which have not been presented before as functional dependencies, the division (relational algebra) operator, SQL hypothetical queries, and Datalog hypothetical queries and rules. Along the paper, for some SQL queries, there are presented equivalent Datalog and RA queries which highlight equivalent formulations is these languages. In this paper we also present for the first time persistence and database interoperability as supported by DES, which allows to test new language features (less-limited SQL recursion, hypothetical queries, division operator, ...) in current relational database systems. Although this is not the first deductive database implementing persistence (cf. [TLLP08, AOT+03]), it develops a seamless integration in a truly interactive system with external DBMS's.

Organization of this paper is as follows. In Section 2, DES is briefly introduced. Next, Section 3 describes the various scenarios for accessing external DBMS's and mixed query solving in DES. Section 4 focuses mainly on the limitations of state-of-the-art SQL-based DBMS's, which can be overcome with a deductive database, also illustrating equivalent formulations in Datalog and RA, and introducing hypothetical queries as an addendum from deductive databases. Although DES is not geared towards performance, Section 5 briefly analyzes some benchmarks that illustrate that a competitive system might be achieved. Finally, Section 6 concludes and points out some future work.

2 Datalog Educational System

DES is a free, open-source, interactive, multiplatform, portable, Prolog-based implementation of a deductive database system. DES 3.2 [SP13] is the current implementation, which enjoys Datalog, (extended) Relational Algebra (RA) and SQL query languages, persistence, full recursive evaluation with tabling, full-fledged arithmetic, strong constraints, stratified negation as described in [Ull88] with safety checks [Ull88, ZCF+97], ODBC connections, and novel approaches to Datalog and SQL declarative debugging [CGS12, CGS11, CGS08], test case generation for SQL views [CGS10], duplicates, null values and outer join support [SP12a], aggregate predicates and functions [SP12b], and hypothetical queries and rules. It is a live system experiencing many downloads (>45K) and used all over the world both for teaching and research (cf. DES Facts at [SP13], and Quotes for feedback from students and teachers).

Interacting with DES is possible via either an OS command shell or GUI applications, as the Java-based IDE ACIDE [SP07], Crimson Editor, Emacs and others. As the system is implemented on top of Prolog, it can be run from a state-of-the-art Prolog interpreter (currently, last versions of SWI-Prolog and SICStus Prolog are supported) on any OS supported by such Prolog interpreter. Portable executables (i.e., they do not need installation and can be run from any directory they are stored) have been also provided for Windows, Linux, and Mac OS X.

There is available a wide set of commands for dealing with the system (in-memory database, ODBC connections, debugging and test case generation, OS interaction, persistence, etc.) They are preceded by a slash to isolate command execution from query solving. Assertions are also provided, as, e.g., type constraints, so that one can type Datalog predicates and make them available for SQL to be queried (whereas Datalog is not a strong typed language, SQL is).

Datalog as supported by DES mainly follows Prolog ISO standard [ISO00] (considering its syntax as a subset of Prolog), whilst SQL follows SQL:2008 ISO standard [ISO08], and RA follows [Cod72] extended with recursion, nulls, outer joins and aggregates (syntax is borrowed from [Die01]). Their concrete syntax can be found at [SP13]. All of these query languages can access the very same database, which is implemented with both an in-memory Prolog database and external relational databases (cf. Section 3).

Evaluation of Datalog recursive 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), ...`}). Note, however, that a Datalog novel top-N query can be submitted, analogously to some current DBMS's, as `top(10,p(X))`, which does terminate.

Datalog temporary views allow 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, the temporary view `d(X) :- a(X), not(b(X))` computes the set difference between the instance relations `a` and `b`.

As it will be explained in the next section, SQL statements can be either compiled to Datalog programs and executed by the deductive engine, or sent to and solved by an external DBMS. Also, SQL statements can be the result of compiling Datalog persistent predicates in order to be externally solved by a DBMS.

3 Enabling Interoperability

Figure 1 depicts the system architecture of DES supporting this functionality. Datalog queries are solved by the deductive engine relying on a cache to store results from fixpoint computations by using a tabling technique [Die87]. The deductive engine is able to solve such queries and to pass SQL queries (as input by the user or as a result of predicate persistence) to external DBMS's. Next, the different scenarios for in-memory, external, and mixed query solving are presented.

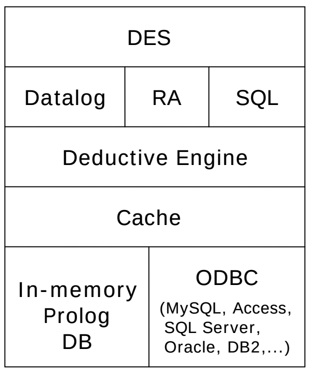

Figure 1: System Architecture

3.1 In-Memory Database

First alternative for dealing with SQL statements is to use the very same Prolog database. SQL row-returning queries for the in-memory database are compiled to and executed as Datalog programs by the deductive engine (basics can be found in [Ull88]), and relational metadata for DDL (Data Definition Language) statements are kept. Submitting a DML (Data Manipulation Language) row-returning 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 and including the compiled Datalog form corresponding to 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 Datalog program and metadata do persist. This allows Datalog programs to seamlessly use tables and views created at the SQL side. Note that SQL metadata (column names and types) are stored as Datalog assertions, and consistency constraints as primary keys and referential integrity constrains are also supported as Datalog assertions.

As an example of a DDL statement, let's consider:

```
create table employee(name varchar, dept varchar, salary integer);
```

which is equivalent (and can be interchangeably used) to the following Datalog assertion (which is denoted by a preceding neck symbol `:-`):

```
:-type(employee(name:varchar, dept:varchar, salary:integer)).
```

Also, integrity constraints can be stated, as the following primary key assertion (or added as a either a column or table constraint in the former `create table` statement):

```
:-pk(employee,[name]).
```

As the result of submitting the former inputs, we can inspect metadata for the current database (here, the in-memory, default database `$des`) with:

```
DES> /db_schema
Info: Database '$des'
Info: Table(s):
 * employee(name:string(varchar),
    dept:string(varchar),salary:number(integer))
    - PK: [name]
Info: No views.
Info: No integrity constraints.
```

Inserting tuples can be done with either an SQL `insert` statement or a command:

```
insert into employee values('Smith','Sales',15000);
/assert employee('Smith','Sales',15000).
```

Selecting tuples can be performed with either an SQL `select`, or a Datalog query, or a RA query. In the following example, employees with a salary over 10,000 are requested for these query languages, resp.:

```
DES> select * from employee where salary > 10000;
answer(employee.name:string(varchar),employee.dept:
string(varchar),employee.salary:number(integer)) ->
{
  answer('Smith','Sales',15000)
}

DES> employee(N,D,S), S>10000
{
  employee('Smith','Sales',15000)
}

DES> select salary>10000 (employee)
answer(employee.name:string(varchar),employee.dept:
string(varchar),employee.salary:number(integer)) ->
{
  answer('Smith','Sales',15000)
}
```

As DES Datalog implementation follows Prolog syntax, variable names start with uppercase (as `N`, `D`, and `S` in the query) and user identifiers start with lowercase (as `employee`). This also applies to RA variable and user identifier names.

Although this in-memory approach is not truly understood as a persistent database, it is still possible to save the contents of the current database to a file and afterwards recover them with the commands `/save_ddb`, and `/restore_ddb`, resp. Next two sections deal with other approaches relying on external DBMS's.

3.2 Connecting to External Databases

An ODBC connection is identified by a name defined at the OS level, and opening a connection in DES means to make it the current database. Any relation defined in the external DBMS as a view or table is allowed as any other relation (predicate) in the deductive database. So, this second alternative to dealing with SQL statements is to use the ODBC bridge to access external databases: Each SQL relation (table or view) is understood as a predicate and therefore can be seamlessly accessed from either a Datalog or a RA query. Contents of SQL relations are retrieved (possibly involving query processing in the case of views) from the current open external DBMS, and SQL queries are directly sent to and processed by such DBMS. As both Datalog and RA queries can refer to SQL relations, therefore, computing such a query can involve computations both in the deductive inference engine and in the external SQL engine. For instance, let's consider the external table `manager(mgr varchar, emp varchar)` stating that `mgr` is the direct manager of `emp`. The following Datalog query computes all managers (both direct and indirect):

```
DES> /open_db mysql
DES> create table manager(mgr varchar(10), emp varchar(10));
DES> insert into manager values ('E1', 'E2'), ('E2', 'E3'), ('E2', 'E4');
DES> managers(M,E) :- manager(M,E) ;
                      manager(M,M2), managers(M2,E).
Info: Processing:
  managers(M,E)
in the program context of the exploded query:
  managers(M,E) :- manager(M,E).
  managers(M,E) :- manager(M,M2), managers(M2,E).
{
  managers('E1','E2'), managers('E1','E3'), managers('E1','E4'),
  managers('E2','E3'), managers('E2','E4')
}
```

This is a Datalog temporary view which recursively computes the outcome as the union (expressed with the semicolon) of direct managers `manager(M,E)` and managers which has also other managers `manager(M,M2), managers(M2,E)`. Note also that the system informs about how this is translated to a query (`managers(M,E)`) and a set of rules (exploded query). Whereas the Datalog engine is responsible of computing the transitive closure, the external SQL engine provides the data source for the relation `manager`. Note that in this example the external computation is rather light, but consider that this relation can be a complex view or we are looking for a concrete employee managers. In this last case, e.g., the submitted SQL query to the external engine would include the `where` condition.

Concluding, opening an ODBC connection allows the user to integrate external relations in Datalog queries, but in this setting it is not possible to integrate Datalog predicates in external SQL queries, as the external database is not aware of Datalog relations. To make it possible, another alternative is presented next.

3.3 Persisting Predicates

Persisting a predicate in DES allows durability for deductive programs by relying on current relational DBMS's as persistent media. We have proposed an assertion as a basic declaration for making a predicate to persist, similar to [CGC+04] (which is for a Prolog system and moreover is not able to persist intensional predicates, but only facts, as DES does allow). The general form of a persistence assertion at the command prompt is as follows:

```
:- persistent(PredSpec[,Connection]).
```

where `persistent` is the keyword for enabling persistence, *`PredSpec`* is a predicate schema specification and the optional argument *`Connection`* is an ODBC connection identifier. Such schema specification can be either *`PredName`*/*`Arity`* or *`PredName`*(*`Schema`*), where *`Schema`* can be either $ArgName_1$, ..., $ArgName_n$ or $ArgName_1$:$Type_1$, ..., $ArgName_n$:$Type_n$. If a connection name is not provided, the current open database is used (the local, default database `$des` cannot be used to persist). With this assertion, we allow for: First, persisting both an empty predicate (i.e., with no defining rules) and an already defined predicates (with defining rules). And, second, for persisting both an untyped or already typed predicate.

The following example makes persistent the predicate `employee`, as already defined in the deductive database in the previous subsection. The second argument (`mysql`) is the name of the ODBC connection (in this case, MySQL is the target DBMS) to map the predicate.

```
:- persistent(employee/3,mysql).
```

Any rule belonging to the definition of a predicate `p` which is being made persistent is expected, in general, to involve calls to other predicates. Each callee (such other called predicate) can be:

- An existing relation in the external database.
- An already persisted predicate which is loaded in the local database.

- An already persisted predicate which is not yet loaded in the local database.
- A predicate which has not been made persistent yet.

For the first two cases, besides making `p` persistent, nothing else is performed when processing its persistence assertion. For the third case, a persistent predicate is automatically restored in the local database, i.e., it is made available to the deductive engine. For the fourth case, each non-persistent predicate is automatically made persistent if types match; otherwise, an error is raised. This is needed in order for the external database to be aware of a predicate which is only known by the deductive engine so far, as this database will eventually compute the meaning of `p`.

However, not all persistent rules are processed by the external DBMS because it does not support some features, and the translations of some built-ins are not supported yet. In the current state of the implementation, the following conditions must hold for a rule to be externally processed:

- The rule does not contain calls to built-ins but comparison operators.
- The rule is not included in a recursive cycle.

Anyway, such rules are stored as metadata information for their persistence. But this does not mean a loss of expressiveness as such are also kept in the in-memory database for computing the meaning of the predicate when needed. This is performed by the deductive engine, which couples the processing of the external database with its own processing to derive the meaning of the predicate. Therefore, all the deductive computing power is preserved although the external persistent media lacks some features as, for instance, recursion. Further releases might contain relaxed conditions for this compilation stage as for, e.g., allowing to project those recursive SQL queries that are supported by the external DBMS.

4 Extending DBMS Expressiveness

This section shows some limits of both expressiveness types (theoretical and practical) in current DBMS's and which are otherwise overcome with DES. All (SQL, Datalog and RA) queries have been actually computed in this system. This, coupled with interoperability, can allow users to test new features with their DBMS of choice.

4.1 Overcoming Contrived Formulations

Some parsers require to write extended formulations even when it would not be strictly needed. For instance, selecting all arguments from an autojoin is not possible in Access and MySQL as:

```
select * from t,t;
```

and table renamings must be added (MySQL requires each derived table to have its own alias).

Some DBMS's as Sybase, MySQL and Access do not include all the outer join versions. Let's consider a full outer join:

```
select * from s full join q on s.sno=q.sno;
```

Therefore, to compute a full outer join one must resort to add more code, as, e.g.:

```
select * from s left join q on s.sno=q.sno
union all
select * from s right join q on s.sno=q.sno;
```

If relations `s` and `q` are queries instead relations, this becomes worse as one has to either duplicate code for them or create new views (with associated issues as changing the database, administration privileges, and hiding original queries).

Also, many nested uses of outer joins are rejected for a number of reasons (including issues related to univocally identify relations and columns in correlations) in different DBMS's (Access, DB2, MySQL, Oracle, PostgreSQL, SQL Server and Sybase) such as:

```
select * from s left join
(select * from q right join sp on q.sno=sp.sno where q.qname<sp.pname)
on s.sno=q.sno where s.name=q.qname;
```

Finally, the absence of operators as `EXCEPT` (cf. MySQL, Access) makes the formulation of several queries more cumbersome and less efficient. However, DES does support it and allow all the above neater formulations.

4.2 Recursive SQL

Let's consider a classical transitive closure problem: Given a graph defined by the relation `edge(ori,des)`, find all paths. A possible, simple, recursive SQL formulation follows:

```
with recursive path(ori,des) as
  select edge.*,1 from edge
union
  select path.ori,edge.des from path,edge where path.des=edge.ori
select * from path;
```

But it is rejected in DB2, Oracle, PostgreSQL and SQL Server because they disallow the `recursive` keyword and/or require `union all`, and parentheses around the local view definition (in turn, neither Access nor MySQL supports SQL recursion). In fact, `union all` is a requisite in standard SQL because of discarding duplicates. However, DES does not require this and both a set or multiset answer can be requested. Also, thanks to the support of Datalog, the following neater formulation is allowed:

```
path(Ori,Des) :-
  edge(Ori,Des)
  ;
  edge(Ori,Int), path(Int,Des).
```

Note that, as the former SQL `with` formulation, this Datalog rule is also a query (a temporary view) which can be submitted at the command prompt. In contrast to current DBMS's, this SQL query can be included in a simplified view declaration in DES as follows, avoiding the use of the `with` clause and additional relation names:

```
create view path(ori,des) as
  select edge.* from edge
union
  select path.ori,edge.des from path,edge where path.des=edge.ori;
```

Another problem with this SQL query in current DBMS's is when the graph includes a recursive cycle. Oracle simply rejects such queries even when they can be actually computed, PostgreSQL enters an infinite loop, and DB2 shows top-500 results by default (requiring all tuples also yields to non-termination). Note that DES, even when enabling duplicates, is able to terminate (duplicate sources are both extensional and intensional definitions [SP12a]) because, in contrast to those systems, checks termination along fixpoint computation.

Current DBMS's show another limitation: Linear recursion is required, which means that only one recursive call is allowed in a recursive definition. The following system session shows, first, how relations `edge` and `path` are made persistent and, second, that non-linear recursion is allowed in DES. To this end, processing of the recursive, non-linear rule is conducted by the deductive engine.

```
:-persistent(edge(a:int,b:int),mysql).
:-persistent(path(a:int,b:int),mysql).
with recursive path(a, b) as
  select * from edge
  union
  select p1.a,p2.b from path p1, path p2 where p1.b=p2.a
select * from path;
```

This example follows previous ones and, as seen, can be formulated as linear-recursive. However, without non-linearity, several queries cannot be expressed, as for instance some graph algorithms [ZCF+97]. Also, recursive SQL queries involving `EXCEPT`, `NOT IN`, and aggregates are not allowed in current DBMS as termination is neither ensured nor detected. But deductive systems implementing XY-stratification can solve some of them, as LDL++ [AOT+03].

Mutual recursion is appealing to formulate some relations, as the hubs and authorities example or the following classical definition for even and odd relations:

```
with
 even(x) as select 0 union select odd.x+1 from odd,
 odd(x)  as select even.x+1 from even
select top 20 x from odd;
```

Note that `select 0` is a from-less SQL statement as accepted in several DBMS's, which simply returns a tuple as specified in the projection list (in this case: `(0)`). However, mutual recursion is not allowed in current DBMS's but otherwise allowed in DES.

4.3 The Division RA Operator

An overlooked, but important, relational algebra operator is division, as found in the original proposal of Codd [Cod72]. No current DBMS includes this operator, which identifies the attribute values from a relation that match with all of the values from another relation. There are several approaches to solve this kind of queries [McC03], but they resort to cumbersome formulations that can be avoided if this operator was made available. DES includes a novel syntax for allowing the division operator in SQL by extending the form that a relation may take as follows (in BNF):

```
Relation ::= ... | Relation DIVISION Relation
```

Consider the subset of Date's [Dat09] famous SuppliersPartsProjects schema `p(pno, pname, color, weight, city)` and `spj(sno, pno, jno, qty)`, for parts and suppliers providing parts, where `sno` and `pno` stand for supplier and part identifiers, respectively. We are interested in the query [McC03]: "Find the `sno` values of the suppliers that supply all parts of weight equal to 17." This can be easily formulated in DES (as opposed to the lengthly and error-prone equivalent formulations in [McC03]) as:

```
select * from
  select sno,pno from spj
  division
  select pno from p where weight=17;
```

One can compare this neater formulation to those (contrived ones) found in [McC03]. This operator is also available in both RA and Datalog as a novelty, and the very same query can be issued in both languages, resp., as:

```
DES> (project sno,pno (spj)) division (project pno (select weight=17 (p)))
```

```
DES> spj(SNO,PNO,_,_,_) division p(PNO,_,_,17,_)
```

A couple of notes in this example are: First, the division operator in Datalog refers to variable names, instead of column names in a schema. And, second, whereas in RA and SQL, projection is needed to build the appropriate schema of involved operands of the division operator, in Datalog it suffices to denote non-relevant variables as anonymous. This discards those non-relevant variables from the schema of the operands for the division operator.

4.4 Functional Dependencies

Functional dependencies are key concepts in learning normalization, but neither standard SQL nor any current DBMS provide a way to enforcing them. Only DB2 supplies the construction `DETERMINED BY` as either a table or column constraint for data cubes. However, such functional dependency is only used to produce an optimized query plan, but it is not possible to enforce it yet. DES follows the syntax in DB2 allowing its enforcing with the following syntax, where `type_ctr` is a type constraint, and `colf`$_i$ and `colt`$_i$ are column names:

```
colf1 type_ctr determined by colt2                 -- column constraint
(colf1,...,colfnl) determined by (colt1,...,coltm) -- table constraint
```

which is equivalent to the also supported Datalog assertion syntax:

```
:-fd(Relation,[colf1,...,colfn],[colt1,...,coltm])
```

4.5 Hypothetical SQL Queries

As a novel feature, DES includes hypothetical SQL queries (absent in the standard) for solving "what-if" scenarios. Date [Dat09] explains this idea (firstly proposed in [SK80]) which broadly means that data can be assumed for a given query without actually modifying its source relations. DES includes a novel syntax for allowing such assumptions in the form of:

```
assume SQL_stmt1 in Rel1 [, ...,SQL_stmtN in RelN]
SQL_query;
```

which allows to assume the result of `SQL_stmt`$_i$ in `Rel`$_i$ when processing `SQL_query`. This syntax for hypothetical SQL clauses follows the syntax of `with` clauses, but using `assume` instead of `with`. Indeed, while a `with` clause allows to *define* new temporary relations with queries, an `assume` clause allows to *modify* the semantics of already-defined relations.

Referring to the former example about recursive paths, one might be interested in assuming that a given edge (say `(3,1)`) is in the relation `edge` in order to know the new transitive closure of `path` without resorting to update `edge`. Following the syntax above, this can be formulated as follows, which assumes another (extensional) edge in the `path` recursive view as defined before:

```
assume select 3,1 in edge select * from path;
```

Also, intensional, recursive rules can be assumed, as in the following query, which computes the transitive closure of `edge` (the relation `path` is not used):

```
assume
  select e1.ori, e2.des from edge e1, edge e2 where e1.des=e2.ori
in edge
select * from edge;
```

Although the current version of DES restricts the use of hypothetical SQL statements to queries at the top-level, a forthcoming release will relax this restriction and allows to both create views with `assume` and also nested uses of assumptions.

4.6 Hypothetical Datalog Queries and Rules

A recent and novel addition to DES has been hypothetical Datalog queries and rules. This approach is based on an intuitionistic semantics [NSS08, McC88, MG12]. As in the former section, extensional as well as intensional data can be assumed and, in addition, hypothetical rules are allowed. Hence, in a forthcoming release it is expected to compile hypothetical SQL queries to Datalog queries, and SQL hypothetical views to Datalog rules. The syntax of an hypothetical literal is as follows:

```
Rule1 /\ ... /\ RuleN => Goal
```

which means that, assuming that the current database is augmented with the rules `Rulei` ($1 \leq i \leq N$), then `Goal` is computed with respect to the current database which is augmented with these rules. Such query is also understood as a literal in the context of a rule, so that any rule can contain hypothetical goals, as in `a :- b => c`. In turn, any `Rulei` can contain hypothetical goals. Variables in `Rulei` are local to `Rulei` (i.e., they are neither shared with other rules nor the goal). Moreover, a hypothetical literal does neither share variables with other literals nor the head of the rule in which it occurs. Also, assumed rules must be safe (in the sense of the safety conditions in [Ull88]). Borrowing an example from [Bon88], the question ”Which are the students (`S`) which would be eligible to graduate (`grad`/1) if taking (`take`/2) `his` and `lp` was enough to graduate?” can be issued as:

```
(grad(S) :- take(S,his), take(S,lp)) => grad(S')
```

This query amounts to assume a new rule only in the context of solving the goal `grad(S')`. Note that the syntax of this rule follows the syntax of Datalog rules. A logical equivalent query would be: $\exists S' grad(S') \Leftarrow \forall S((take(S,his) \wedge take(S,lp)) \Rightarrow grad(S))$.

5 A Note on Performance

Although DES has not been designed under any performance directive, some numbers can be taken to analyze its behavior w.r.t. other systems. Here, we consider the logic programming system XSB 3.2, as an example of an efficient open-source, in-memory, tabled deductive database, and the system IBM DB2 10.1.0, as a commercial, persistent, relational database system. These systems are compared with the DES in-memory system and also with the persistent approach as described in Section 3.3. In this last case, DB2 is the target system providing persistence via an ODBC connection.

As tests, we consider the cost of computing the Cartesian product of a table with itself, depending on the table size (benchmark instances). We focus therefore in retrieving to the main memory the result but without actually displaying the result in order to elide the display time. Each test has been run 10 times, the maximum and the minimum numbers have been discarded, and then the average has been computed. All benchmarks are given in milliseconds and have been run on an Intel Core2 Quad CPU at 2.4GHz and 3GB RAM, running Windows XP 32bit SP3. DB2 was accessed through the IBM DB2 ODBC driver version 10.01.00.872.

Table 1 collects such time data expressed as milliseconds. Also, the benchmark instance is denoted in column Tuples, which shows the number of tuples in the table. The column Result shows the number of tuples in the result set which will be retrieved to main memory. Columns XSB, DES, DB2, and P-DES show the computation times for retrieving into main memory the result of the Cartesian product for XSB, DES for the in-memory database, DB2, and DES with the table as a persistent predicate mapped to DB2.

Tuples	XSB	DES	DB2	P-DES	Result
200	197	135	360	816	40,000
400	283	557	1,459	3,035	160,000
600	416	1,266	3,270	6,740	360,000
800	572	2,287	5,783	11,912	640,000
1,000	768	3,646	9,100	18,590	1,000,000

Table 1: Performance Data

Figure 2 graphically collects these numbers for the same range of tuples as in the table, and shown in the horizontal axis. In turn, vertical axis shows computation times expressed in milliseconds. The number computed tuples are also represented (which has been scaled down (50 times) to compare with the time to compute them) and it is depicted in the top curve. As illustrated, the ranking of computing times are in this order, from the best to the worst: XSB, DES, DB2, and P-DES.

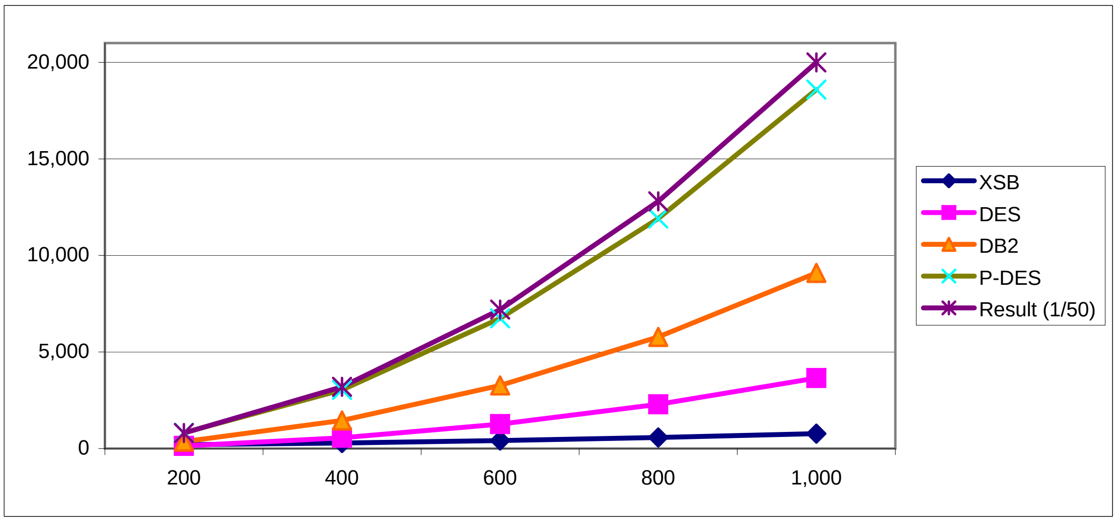

Figure 2: Performance Curves

A first observation from the figures is that the worst case corresponds to DES with persistent predicates, and its curve follows a similar slope to the number of computed tuples, meaning that the time complexity is linear in this number. A second observation is that the curve for the in-memory DES system performs better than DB2. This is clearly expected as DB2 takes more tasks than DES to compute such tuples. For instance, first, tables are not in-memory data structures, but otherwise rely on the OS file system. Nevertheless, it features an efficient buffer manager that tries to keep all data in main memory. And, second, it has to deal with transactional behavior, although not with logging as there are no updates in the tests. Finally, XSB achieves the best numbers as it is a C implementation which compiles the predicates and indexes tabled results with tries.

Therefore, given these numbers, it can be expected to develop an efficient deductive system, in particular as fast as DB2, for solving these tests because the computation time due to the in-memory XSB system is negligible with respect to the persistence requirements for retrieving only 1,000 tuples from the external DBMS. This would be needed to read the table before actually computing the Cartesian product.

6 Conclusions

This paper has shown an ongoing educational project dealing with SQL expressiveness issues and allowing to project computations to external DBMS's. By providing a less-limited parser for practical expressiveness and projecting results from deductive databases for theoretical expressiveness, a better SQL language has been achieved. Although this is not the single system providing such features, it is the only one which includes altogether all the ones presented in this paper. In addition, no other one include outer joins, intensional rule duplicates, SQL functional dependencies, the division operator, and hypothetical queries, where some of these features are reported here for the first time (other interesting features include SQL algorithmic debugging, which are out of the scope of this paper). However, as an educational tool, it cannot compete with state-of-the-art systems w.r.t. performance. So, one of the obvious steps to continue this work is to apply known techniques to enhance performance (tabling trie-based indexing, native compilation, . . .) and even to target to efficient tabled systems as XSB.

Bibliography

[AOT$^+$03] F. Arni, K. Ong, S. Tsur, H. Wang, C. Zaniolo. The Deductive Database System LDL++. *TPLP* 3(1):61–94, 2003.

[Bon88] A. Bonner. Hypothetical Datalog: Complexity and Expressibility. *Theoretical Computer Science* 76:144–160, 1988.

[CGC$^+$04] J. Correas, J. M. Gómez, M. Carro, D. Cabeza, M. V. Hermenegildo. A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations). In Jayaraman (ed.), *PADL*. LNCS 3057, pp. 104–119. Springer, 2004.

[CGS08] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. A Theoretical Framework for the Declarative Debugging of Datalog Programs. In *International Workshop on Semantics in Data and Knowledge Bases*. LNCS 4925, pp. 143–159. Springer, 2008.

[CGS10] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. Applying Constraint Logic Programming to SQL Test Case Generation. In *Proc. International Symposium on Functional and Logic Programming (FLOPS'10)*. LNCS 6009. 2010.

[CGS11] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. Algorithmic Debugging of SQL Views. In *Eigth Ershov Invormatics Conference, PSI'11*. Pp. 204–210. June 2011.

[CGS12] 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'12)*. LNCS 7294. Springer, 2012.

[Cod72] E. Codd. Relational Completeness of Data Base Sublanguages. In Rustin (ed.), *Data base Systems*. Courant Computer Science Symposia Series 6. Englewood Cliffs, N.J. Prentice-Hall, 1972.

[Dat09] C. J. Date. *SQL and relational theory: how to write accurate SQL code*. O'Reilly, Sebastopol, CA, 2009.

[Die87] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *IEEE Symp. on Logic Programming*. Pp. 264–272. 1987.

[Die01] S. Dietrich. *Understanding Relational Database Query Languages*. Prentice Hall, 2001.

[Got12] G. Gottlob. Second Datalog 2.0 Workshop. Austria, Vienna, 2012.

[ISO00] ISO/IEC. ISO/IEC 132111-2: Prolog Standard. 2000.

[ISO08] ISO/IEC. SQL:2008 ISO/IEC 9075(1-4,9-11,13,14):2008 Standard. 2008.

[McC88] L. T. McCarty. Clausal Intuitionistic Logic I - Fixed-Point Semantics. *J. Log. Program.* 5(1):1–31, 1988.

[McC03] L. I. McCann. On Making Relational Division Comprehensible. In *ASEE/IEEE Frontiers in Education Conference*. 2003.

[MG12] D. Miller, N. Gopalan. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[NSS08] S. Nieva, F. Sáenz-Pérez, J. Sánchez. Formalizing a Constraint Deductive Database Language based on Hereditary Harrop Formulas with Negation. In *FLOPS'08, Proceedings*. LNCS 4989, pp. 289–304. Springer-Verlag, Ise, Japan, 2008.

[RU93] R. Ramakrishnan, J. Ullman. A survey of research on Deductive Databases. *JLP* 23(2):125–149, 1993.

[SP07] F. Sáenz-Pérez. ACIDE: An Integrated Development Environment Configurable for LaTeX. *The PracTeX Journal* 2007(3), August 2007. http://acide.sourceforge.net.

[SP12a] F. Sáenz-Pérez. Tabling with Support for Relational Features in a Deductive Database. In *XII Jornadas sobre Programación y Lenguajes (PROLE'12)*. 2012.

[SP12b] F. Sáenz-Pérez. Outer Joins in a Deductive Database System. *Electronic Notes in Theoretical Computer Science* 282(0):73 – 88, 2012.

[SP13] F. Sáenz-Pérez. Datalog Educational System 3.2. February 2013. http://des.sourceforge.net/.

[SCG11] F. Sáenz-Pérez, R. Caballero, Y. García-Ruiz. A Deductive Database with Datalog and SQL Query Languages. In Yang (ed.), *APLAS*. LNCS 7078, pp. 66–73. Springer, 2011.

[SK80] M. Stonebraker, K. Keller. Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System. In *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*. SIGMOD '80, pp. 58–66. ACM, New York, NY, USA, 1980.

[TLLP08] G. Terracina, N. Leone, V. Lio, C. Panetta. Experimenting with recursive queries in database and logic programming systems. *TPLP* 8(2):129–165, Mar. 2008.

[Ull88] J. D. Ullman. *Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1988.

[ZCF$^+$97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.