DEVISER: Datalog EValuator Integrated with SchemE

K. Lisovsky, A. Markov, A. Semenov

September 6, 2005

Abstract

We present programming system DEVISER: Datalog EValuator Integrated with SchemE. DEVISER allows to store tuples in the database, write programs on general-purpose language Scheme and write Datalog programs and query the database in the Scheme program in natural manner. We discuss the design and implementation of our system, and show how functional programming language Scheme contributes to the ease of Datalog language usage.

1 Introduction

Deductive database systems are database management systems whose query language and (usually) storage structure are designed around a logical model of data. As relations are naturally thought of as the "value" of a logical predicate, and relational languages such as SQL are syntactic sugarings of a limited form of logical expression, it is easy to see deductive database systems as an advanced form of relational systems.

Deductive systems are not the only class of systems with a claim to being an extension of relational systems. The deductive systems do however share with the relational systems the important property of being declarative, that is of allowing the user to declare certain application task recursively in terms of universe rather than specify the set of operations for solving that task. Declarativeness is recognized as an important driver of the success of relational systems [1].

We consider Datalog language [2] as the main language for querying database. Datalog is more declarative and closer to databases than Prolog. Datalog as a logic programming language is declarative language, another important kind of declarative programming is functional programming. Logic programming is positioned in the next level abstraction in regard to functional programming. We use functional approach in the implementation. It was interesting in what way concepts of Datalog may be implemented in this approach, how they would look in functional formalism. It was natural to expect that family Datalog and Scheme are rather similar, that their tight integration, i.e. embedding Datalog sentences into base language, is possible. Our contribution is to apply functional methods to the logic programming and database theory. There is not any similar system. The code and documentation of the system is available from http://sp.cmc.msu.ru/datalog

The paper is structured as follows. Section 2 describes the design of the Scheme interface. Section 3 clarifies details of the implementation. Section 4 presents the results of real application of our system. Section 5 reviews related works and Section 6 concludes.

2 Design

DEVISER embeds in Scheme a little language for creating and manipulating queries and for obtaining new results from database by evaluating Datalog programs. Objects of Datalog such as rules, queries, predicates, etc. can be treated as first class citizens, thus considerably raising the level of abstraction programmer can use [3].

The DEVISER's Datalog grammar is very simple.

⟨program⟩ $\longrightarrow$ (⟨rule⟩ ...)
⟨rule⟩ $\longrightarrow$ ⟨predicate list⟩
⟨query⟩ $\longrightarrow$ ⟨predicate list⟩
⟨predicate list⟩ $\longrightarrow$ (⟨predicate⟩ ...)
⟨predicate⟩ $\longrightarrow$ (⟨predicate-symbol⟩ ⟨term⟩ ...)
    | ⟨specific expression⟩
⟨term⟩ $\longrightarrow$ ⟨constant⟩ | ⟨variable⟩
⟨variable⟩ $\longrightarrow$ ⟨identifier⟩ | _
⟨specific expression⟩ $\longrightarrow$ `(dtlog-bpr!` ⟨binding expression⟩`)`
    | `(dtlog-bpr!` ⟨computable expression⟩`)`
⟨computable expression⟩ $\longrightarrow$ (⟨scheme function⟩ ⟨computable expression⟩ ...)
    | `identifier`
    | `constant`
⟨binding expression⟩ $\longrightarrow$ `identifier` ⟨computable expression⟩
    | `identifier` ⟨aggregation function⟩ ⟨predicate list⟩ `identifier`
⟨aggregation function⟩ $\longrightarrow$ `count!` | `sum!` | `max!` | `min!` | `avg!`

Pure Datalog [2] is too poor, it prevents from formulating many real application tasks. Maybe it would be better to maintain Datalog purity analogously to 'pure' standard SQL ideas, but we prefer more realistic approach. We extend Datalog by adding computable built-in predicates, aggregate functions over sets and binding expressions.

Computable expressions can be thought as usual predicates of database. But storing all relations between two integers is extremely unprofitable. Therefore computable expressions are evaluated by performing specified operations (Scheme functions) using instantiated variables. But all computable expressions are under the safety restriction: all contained variables must be instantiated.

When evaluator finds standalone computable expression (dtlog-bpr! computable expression), it evaluates this expression when it is possible and uses evaluated Boolean value. If

⟨computable expression⟩ is #t (Scheme value), this built-in predicate with current instantiated variables becomes true. Otherwise it is false. There is an example of rule with computable expression:

```
(map (lambda (x)
       (list
         (tuple-ref x 0)
         (tuple-ref x 1)
         (tuple-ref x 6)))
     (dtlog-eval-query rules query edb-in))
```

Above s-expression is the Scheme analog to the following rule:

```
in_range(X, A, B) :- int(X), int(A), int(B), A < B, X <= B, X >= A.
```

Binding expression is useful when you want to denote using some variable the whole big computable expression or aggregate function. Computable expression can return all what you want coherent to the whole inference or evaluation process. You can write Fibonacci rule:

```
((fibonacci N F) (dtlog-bpr! (> N 2)) (fibonacci N1 F1) (dtlog-bpr!
eq? N1 (- N 1)) (fibonacci N2 F2) (dtlog-bpr! eq? N2 (- N 2))
(dtlog-bpr! F (+ F1 F2)))
```

The same example in the traditional notation:

```
fibonacci(N,F) :- N > 2, fibonacci(N1,F1), N1 == N - 1,
fibonacci(N2,F2), N2 == N - 2, F = F1 +F2.
```

In the binding expression you can use aggregate function over the sets of facts. The function aggregates the result of immediate evaluation of predicate list with current predicate values. Therefore all variables of these predicate list must be instantiated. Aggregate functions are built in DEVISER, but you can easily write your own function. The last identifier in that binding expression with aggregate function indicates the accumulating or aggregating variable in the predicate list.

```
((avg_salary Dep Avg) (departments Dep) (dtlog-bpr! Avg avg!
((salary _ Dep S)) S))
```

Above s-expression is the Scheme analog to the following rule:

```
avg_salary(Dep, Avg) :- departments(Dep), Avg = avg!([salary(_,
Dep, S)], S).
```

It is easy to write Datalog program using Scheme. Application program interface consists only of few functions `dtlog-db`, `dtlog-rule`, `dtlog-query`, `dtlog-eval`, `dtlog-eval-query`, `dtlog-print-result`. There is their signature:

(`dtlog-db` *filename*)

Function `dtlog-db` takes as input the name of file containing extensional database under the management of SQLite dbms. Function returns so-called "connection" with database. Further function calls take it as input to extract data.

(`dtlog-rule` *predicate-list*)
(`dtlog-query` *predicate-list*)

This is a pair of crucial functions. They work as the grammar analyzer. As input these functions take the list of atoms. If function `dtlog-rule` is used, it is assumed that the first atom is a head of rule's body. Functions `dtlog-rule` and `dtlog-query` returns internal representation of rule and query to program respectively.

(`dtlog-eval` *program db*)
(`dtlog-eval-query` *program query db*)

Functions `dtlog-eval` and `dtlog-eval-query` evaluate Datalog program by semi-naive bottom-up method and query subquery (qsq) method respectively. Both functions take as input the list of rules. Every rule in these list must be obtained by `dtlog-rule` call. Function `dtlog-eval-query` has another argument, obtained by function `dtlog-query` call. Both functions return whole evaluated result. The result is the list of facts which can be printed by function `dtlog-print-result`.

(`dtlog-print-result` *idb*)

The main functions of Scheme API are `dtlog-rule` and `dtlog-query`. They are written with help of R5RS macros. It is possible to use any Scheme-expressions in appropriate Datalog constructions. Binding of variables is made by Datalog's principles, not by Scheme principles. It means that you can write some Scheme functions of n arguments. Then in the evaluation process of Datalog program each time when all variables in the expression are instantiated, this Scheme function will be applied to these values of variables.

3 Implementation

When we say 'coupling', we mean implementation of interface between two various subsystems, logic programming system and dbms. Both systems preserve their independence, there is an interface to transmit data from database to main memory of Datalog evaluator. This approach is the simplest, but isn't very efficient. We use exactly this approach because developing and implementation of database management system with reliable, permanent and efficient data storage. Using Datalog as a query language is very complex.

Tight coupling is a kind of coupling when there is permanent interaction between logic subsystem and dbms. Every time when new facts are needed by logic inference, corresponding query to dbms is used. Loosely coupling characterized by loading all potentially relevant facts from database to main memory of logic subsystem before starting of evaluating logic program.

We use loose coupling. It consumes more memory, but we think that developing tightly coupled systems involves tight integration with specific dbms and perhaps it involves dependence on that dbms and impossibility of refusing and changing it.

At early stages of our project we use PLT Scheme [4]. But the following facts forced us to use Bigloo Scheme [5]:

- at the moment there isn't any fast portable interface with dbms, which would be standard for Scheme;
- Bigloo Scheme has interface to various dbms;
- Bigloo Scheme is actually the fastest Scheme R5RS compiler.

Bigloo translates Scheme program to C, Java and MSIL. So you may create executables for various platforms including Unix and Windows. But we have only R5RS code, dependence on the Bigloo is appeared only in the interaction with dbms.

We use SQLite dbms [6], because it is less complicated, fast, and portable and it has a simple API. For open research project it was the best choice.

Due to Bigloo capabilities DEVISER exists as standalone system too. A user can write programs as text files using special grammar, give this text program and a SQLite database as an input to DEVISER executable, and obtain result. Details can be found at [7]. All above examples are written in Datalog, not in the presented Datalog in Scheme, are valid parts of text programs.

4 DEVISER in action

Let's consider DEVISER in action. We demonstrate our results as a practical illustration of DEVISER's work. We investigate two aspects of real applicability: performance and expressive capabilities.

4.1 Performance

At the moment DEVISER has four calculating methods. There is three bottom-up methods (Jacobi, Gauss-Seidel and generalized semi-naive) and one top-down method query subquery (qsq).

In all benchmarks various extensional databases are used but these databases are created by common principle. Fact is a pair of two random numbers from 1 to Max Facts constant. This

N	Total amount of facts		Measured time (seconds)			
	Initial	Inferred	Jacobi	Gauss-S	s-i	qsq
2	80	127	0.03	0.05	0.03	0.04
6	240	930	2.37	2.33	0.95	2.11
8	320	3818	52.83	53.07	20.33	51.56
10	400	12643	601.53	594.41	209.37	575.98
11	440	32864	-	-	1602.66	-
12	600	59524	-	-	3528	-

Table 1: Evaluation of program L1. Max Facts = 90000

constant defines maximum possible number of facts in the binary predicate. We present results of the most interesting benchmark.

Measurements are done under Linux RedHat 9.0 at Intel Pentium IV 1.5 Ghz, 256 Mb RAM.

Program L1:

```
(dtlog-eval `(,(dtlog-rule ((anc X Y) (anc X Z) (par Z Y))
                ,(dtlog-rule ((anc X Y) (par X Y))) edb-in)
```

Program L1 demonstrates calculating capabilities of DEVISER. Results are presented in the Table 1. Method qsq was applied to the query `((anc X Y))`. Sign '-' denotes operation system message "Out of memory" after long computations.

Semi-naive method has demonstrated acceptable performance time with inferred facts up to 5000. The authors think presented results confirm implementation's success and possibility of real application of DEVISER to the databases with reasonable size. Readers should keep in mind that special optimization of Scheme code has not been made.

4.2 Expressiveness

For demonstration of expressive capabilities antitrust problem from [2] is used. Let's briefly describe the statement.

There are many companies which make goods. Goods are sold on specified markets. Each company possesses the certain quota of market. Company can also control quota of market if it has more than 50 % of stocks of another company, which has its own part of market. Many countries use economic regulation, which does not allow the companies to control explicitly or implicitly the quota of market more than specified. Companies constitute a trust when they break the law.

We have 3 extensional predicates `has_shares(company1, company2, control)`, `company(company, market, quota)`, `trust_limit(market, limit)`. The first describes the relation that `company1` has `control` % of stocks of `company2`. The second denotes that `company` has `quota` % of `market`. The third describes that `market` has `limit` %. We need to know which of given companies lead the trusts and what does quota they have.

The following Scheme program solves this problem with given edb filename of the database:

```
(define edb-in (dtlog-edb "trust.sqlite"))

(define rules `(
  ,(dtlog-rule
     ((controlled Comp2 Comp1)
     (has_shares Comp1 Comp2 N) (dtlog-evpr! (> N 50))))
  ,(dtlog-rule
     ((all_controls Comp1 Comp2)
     (controlled Comp2 Comp1)))
  ,(dtlog-rule
     ((all_controls Comp1 Comp2)
     (all_controls Comp1 Comp3) (controlled Comp2 Comp3)))))

(define query
  (dtlog-query ((company Comp1 Market Quota1)
    (dtlog-evpr! Total1 sum! ((all_controls Comp1 Comp2)
      (company Comp2 Market Quota2))  Quota2)
    (dtlog-evpr! Total (+ Total1 Quota1))
      (trust_limit Market Threshold)
    (dtlog-evpr! (> Total Threshold)))))
```

Let EDB predicates contain following values:

```
has_shares = '((violet hyachint 51) (violet orchid 20) (violet
    gardenia 30) (violet lily 51) (hyachint daisy 30) (hyachint iris 70)
    (gardenia orchid 51) (fiat opel 2) (gardenia tulip 60) (lily rose
    59) (tulip azalea 20) (tulip begonia 25) (begonia azalea 60))
company = '((violet toys 8) (hyachint toys 7) (opel cars 33) (orchid
    toys 13) (gardenia toys 0) (tulip toys 12) (ford cars 30) (daisy
    toys 8) (iris toys 12) (lily toys 2) (rose toys 4) (azalea toys 19)
    (begonia toys 15) (fiat cars 35))
trust_limit = '((toys 20) (cars 38) (gasoline 40))
```

Then simple Scheme expression

```
(map
  (lambda (x)
    (list
      (tuple-ref x 0)
      (tuple-ref x 1)
      (tuple-ref x 6)))
  (dtlog-eval-query rules query edb-in))
```

returns the result:

```
'((violet toys 33) (gardenia toys 25) (begonia toys 34))
```

Companies violet, gardenia and begonia form trusts on the toy market with actual quotas 33, 25 and 34 respectively.

5 Previous Related Work

Let's briefly observe existent similar systems and note the position of DEVISER among them. Survey is based on [1], [8]. In many respects DEVISER is designed under the influence of the fact that there had not been any Datalog's evaluator written at Scheme. Thus one might say DEVISER is the first system based on Datalog and is written in Scheme and integrated with Scheme. It is important because Scheme is very popular now in academic and research community. Therefore DEVISER itself is research project, may be used in much others researches and in many others research projects.

There are many logical programming systems, but most of them are based on Prolog language. One of the most famous is XSB. There are variety Prolog's evaluators written in Scheme. One of the latest and the most interesting is Kanren [9].

There are only few systems based on Datalog. Comparison is made with DLV [10] and LDL++ [11]. There are common features in these projects:

- these systems exist for a long time, contain implementation of many various methods concerning Datalog: rewriting, optimization, negation, aggregation and others;
- these systems are written in traditional industrial programming languages, so in the performance aspect DEVISER loses to these systems at the least 10 times at small databases.
- for research purposes you may freely use these systems, but source codes are inaccessible.

The lack of the good performance results or variety implemented methods applied to Datalog is a consequence of limitations of our investigation. The presented work is student research project, implemented by A. Semenov on System Programming Chair of Computational Mathematics and Cybernetics department of Moscow State Univercity.

But in contrast to considered systems with good performance and capabilities, DEVISER is an open system. It is an advantage in using DEVISER and it should attract researchers in their investigations.

6 Conclusion

We have presented Datalog EValuator Integrated with SchemE. DEVISER involves the ideas of logic programming, database theory and functional programming. It allows natural embedding of Datalog programs to the Scheme programs, using Scheme functions during the evaluation of Datalog program and storing facts in the database management system.

We have demonstrated viability of our approach by performance benchmarks and useful Scheme interface.

Future work assumes following code optimization, careful testing and benchmarking, concurrent algorithmization, solving application problems. Welcome interested readers to our page http://sp.cmc.msu.ru/datalog.

References

[1] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[2] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer, 1990.

[3] N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two Little Languages, 2002.

[4] PLT Scheme Homepage. http://www.plt-scheme.org/.

[5] Bigloo Scheme Homepage. http://www-sop.inria.fr/mimosa/fp/Bigloo.

[6] SQLite Homepage. http://www.sqlite.org/.

[7] DEVISER Homepage. http://sp.cmc.msu.ru/datalog.

[8] C. Zaniolo. *The Handbook of Data Mining and Knowledge Discovery*, chapter A Short Overview of Deductive Database Systems. Oxford University Press, 1999.

[9] Daniel Friedman and Oleg Kiselyov. A Logic System with First-Class Relations. Unpublished manuscript, http://www.cs.indiana.edu/l/www/classes/b521/qs.ps, May 2004.

[10] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Georg Gottlob, Giovambattista Ianni, Giuseppe Ielpa, Christoph Koch, Simona Perri, and Axel Polleres. The DLV System. In *JELIA*, pages 537–540, 2002.

[11] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. The Deductive Database System LDL++. *TPLP*, 3(1):61–94, 2003.