Effective Datalog-like representation of procedural programs[⋆]

David Bednárek

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University Prague
bednarek@ksi.mff.cuni.cz

Abstract. *Database systems are constantly extending their application area towards more general computing. However, applications that combine database access and general computing still suffer from the conceptual and technical gap between relational algebra and procedural programming. In this paper, we show that procedural programs may be effectively represented in a Datalog-like language with functions and aggregates. Such a language may then be used as a common representation for both relational and procedural part of an application.*

1 Introduction

As database systems were extending their application area, their mainstay language, SQL, became no longer sufficient to support all required applications of databases. The database community reacted in two ways: Implementing domain specific languages like XQuery, SPARQL etc. and improving the interaction of database engines with general procedural programming languages. The second approach is certainly more general but it is also more difficult due to deep differences between the procedural and relational paradigms.

Many database systems offer a procedural language with embedded SQL statements. The most common processing scenario is depicted at Fig. 1. The procedural and relational parts are separated already in the parser stage. The procedural part is converted to a procedural intermediate representation while the relational part is expressed using extended relational algebra. The two parts are processed almost independently in the following stages. When entering run time, the procedural part is usually expressed by a procedural bytecode which is designed for easy interpretation; the relational part is represented by a physical plan, i.e. an expression over a physical algebra.

Database-related optimizations are performed only on the relational side; individual SQL statements extracted from the source code are usually optimized independently. The most important optimization step is the strategy of physical plan generation, traditionally called *cost-based optimization.* Here, the logical relational algebra operators are converted to physical

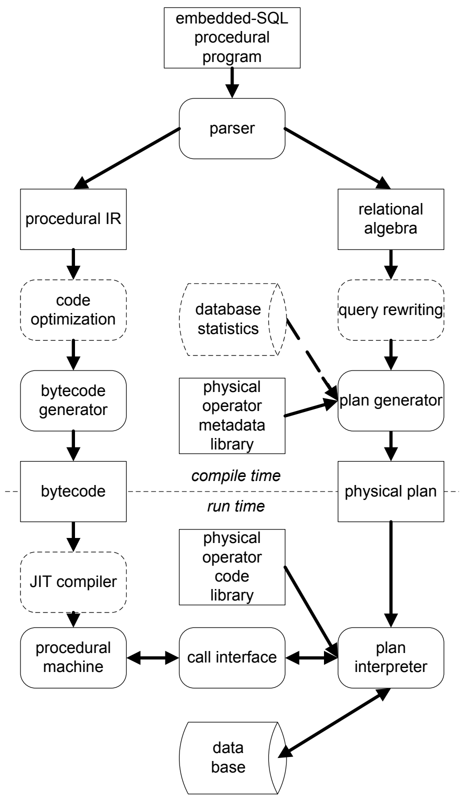

Fig. 1. Typical processing path of embedded-SQL programs

operators selected from a library and the selection is guided by database statistics and physical operator metadata (cost etc.).

The procedural part meets with its relational elements only at run time – the procedural machine executes the procedural code containing calls to a database interface which in turn invokes the plan interpreter. The interpreter schedules and dispatches calls to procedures implementing individual physical operators of a plan. Although the individual SQL statements extracted from the source program often access the same data, their plans usually interact only at the lowest levels of physical data access; thus, they often repeat the same operations on the same data.

⋆ The work was supported by the GACR project 202/10/0761 and by the GAUK grant SVV-2011-263312.

The most important reason for the separation of procedural and relational parts is probably the history – the relational (SQL) query engines were usually implemented well before the procedural *add-on* was considered and the software-engineering cost of reimplementing the query engine was considered too high.

Nevertheless, there is another reason for the separation – the nature of procedural code is so distant from the algebraic nature of SQL that it is very difficult to create a meaningful common representation for the two parts.

1.1 Goals

In this paper, we suggest using a Datalog-like language as the common intermediate representation for procedural and relational code. This idea is natural, considering the close relationship between Datalog and relational algebra on one side and the computing power of logic programming and recursion on the other side. However, there are still many obstacles in the integration effort. In particular, there is a risk of loss of efficiency since the procedural code is not evaluated directly but transformed to logic-programming representation and then evaluated by a procedural hardware.

In our approach, we strive to improve the efficiency of logic-programming representation by minimizing the size of the models – we show that a procedural code can be encoded in a logic program in such a manner that the size of the (minimal) model is proportional to the execution time of the original program on the same data. Thanks to this proportionality, the logic program may be evaluated completely in bottom-up manner.

Due to the focus on bottom-up evaluation, we prefer calling our approach *Datalog-like* over the term *logic programming*, although we certainly must use a language stronger than Datalog to achieve generality.

Besides function symbols which are necessary to simulate general procedural programming, our language needs negation and aggregation for the emulation of both procedural constructs and the embedded relational language. Both negation and aggregation require special handling to ensure well-defined semantics and there are several approaches to the semantics of negation in Datalog and logic programming in general.

Thus, defining a particular approach to semantics is a part of our effort – we reuse the concepts of *local stratification* [13] and *rule progressivity* [10] that together well reflect the nature of the original procedural code.

1.2 Architecture

The architecture of the proposed system is shown at Fig. 2. Instead of separating the procedural and relational part, the parser converts the source code into a Datalog-like intermediate representation. This intermediate language and the conversion of procedural code into it form the main subject of this paper. On the other hand, conversion of relational queries to Datalog is a well-known subject; thus, it is not necessary to describe the handling of embedded SQL statements.

The processing continues with rewriting step which optimizes the Datalog-like representation – this phase corresponds to query rewriting and it may also be influenced by database statistics. Moreover, several optimization algorithms known from compiler construction like loop unrolling or variable renaming [17] have their equivalents in the transformation of logic programs, so this phase covers also the optimization of procedural code.

The most important feature of this architecture is the ability to apply rewriting optimization step across the boundary between procedural and relational code. For instance, repeated invocations of a SQL statement may be glued together, offering, for instance, the possibility to cache their partial results.

The plan generator phase tries to cover the Datalog-like representation using a predefined set of patterns. Each pattern corresponds to a *component* which has several inputs and several outputs, each corresponding to a predicate. A simplest component cover one Datalog rule – in this case, the head of the rule corresponds to the output and each atom of the body corresponds to an input of the component. More sophisticated components correspond to a pattern covering more than one rule, including recursively dependent rules.

Some of the components correspond to physical operators of a relational engine; for instance, a Datalog rule with two body atoms may be implemented by a hash-join operator. Other components are implemented with simple procedural code snippets – these components ensure that the procedural parts of the source code are reverted back to procedural code. Of course, parts of the source code may be changed during the rewriting phase; consequently, procedural source code may be eventually covered by relational operators and, conversely, portions of the embedded relational statements may be converted to procedural snippets.

The implementation of physical relational operators is boxed in procedural packages which are connected together similarly to classical query plans. On the other hand, the procedural snippets are so small that individual packaging would be ineffective. Therefore, the snippets are combined together to larger procedural code fragments by the bytecode generator.

This paper deals with the design of the intermediate representation and the translation from procedural code to it. The subsequent steps – Datalog-based rewriting and plan generator – are subject of our current research. The run-time portion of the system at Fig. 2 was already implemented for a SPARQL compiler [4] whose compile time used a relational intermediate code and an algebra-based plan generator.

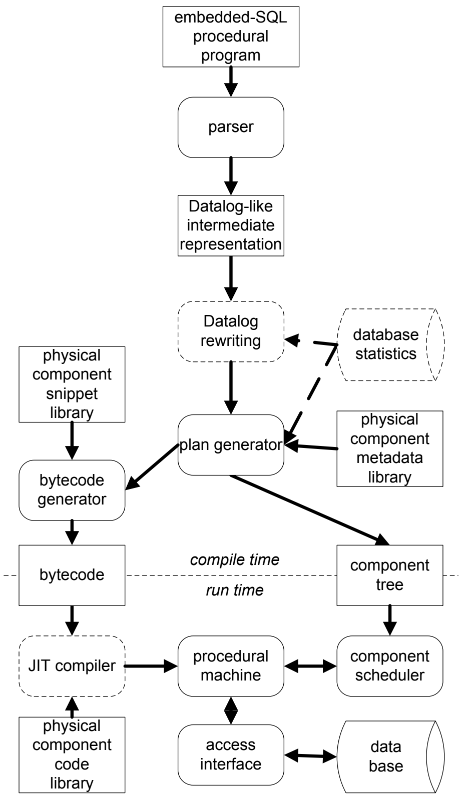

Fig. 2. Proposed processing path of embedded-SQL programs.

The rest of the paper is organized as follows: In the Section 2, we briefly review the related work on the interaction between procedural and relational code as well as Datalog-related definitions important for our approach. The following section uses a sample procedural code to illustrate several approaches to the modelling of procedural code in Datalog style. After reviewing the required extensions to Datalog in Section 3.5, a particular strategy to the minimization of model size is presented in Section 3.6.

2 Background and related work

Interaction of procedural and relational code was recognized as an important topic a long time ago; nevertheless, practical applications of the results are very scarce. In [7], a successful optimization of the interaction cost was shown in the case of calling procedurally-implemented functions from relational queries. Our approach also applies to this case; nevertheless, we focused on the opposite problem – repeated calling of relational queries from procedural code.

Nested relational algebra [14] may well reflect the use of structured and relation-valued variables in a procedural program; however, the overall computing strength of nested relational algebra is insufficient to express while loops. While loops may be added as an additional second-order construct atop of relational algebra, or represented by transitive closure in powerset algebra [9]. In Sec. 3.3, we will show a logical-programming equivalent of nested relational algebra together with the drawbacks of such an approach.

Flattening nested relations is an important step towards effective evaluation of nested algebra and it is also present in the core of our approach. The original flattening principle described in [16] was designed to flatten an isolated nested-relational algebra expression and it was based on the finite height of the expression tree. Since a while loop may generate a calculation of unlimited length, this flattening technique may not be applied for procedural programs. Instead, we had to use a numbering technique (see Sec. 3.4) and to solve some unwanted consequences of the numbering approach.

Datalog and its extensions, besides their natural applications in many areas of database theory, was already successfully used in areas related to procedural programming.

A language derived from logical programming was designed for programming in distributed environment [5]. The opposite problem, generating effective procedural implementations from Datalog programs, was studied in [11]. These recent publications suggest that the potential of Datalog was not exhausted in the first decades of its life and it may experience a revival fueled by the renewed interest in non-traditional database architectures.

There were attempts to improve the expressive power of Datalog towards procedural programming by non-traditional extensions of its semantics [8]. Extending Datalog towards complex data structures known from procedural programming was described in [6]. These powerful extensions relied on significant intrusions to the traditional Datalog semantics; consequently, their use in an intermediate language for a relational platform would be doubtful.

Datalog with temporal features was used to model sequences of data-manipulation statements [10] or in the analysis of procedural programs [15]. Our numbering technique shown in Sec. 3.4 is similar to temporal techniques although used for a different purpose.

Among the sheer number of approaches to Datalog extensions, semantics and evaluation strategies, *local* and, in particular, *temporal stratification* [12] has motivated our approach. In addition, we also make use of the *progressivity* [10] of rules in the scheduling strategy used in plan execution.

3 Modeling procedural code in Datalog style

Algorithm 1 Example: A procedural algorithm with embedded SQL statements

```
Require: Relation $M(A, B)$
1: $S := z$
2: while exists($M$) do
3:     $X$ := select min($A$) from $M$
           where $A$ not in (select $B$ from $M$)
4:     $S := f(S, X)$
5:     delete from $M$ where $A = X$
6: end while
7: return $S$
```

Algorithm 1 is an example of code written in a procedural language with SQL embedded. It consumes a relation M with schema (A, B) representing edges of a directed graph and traverses it in a topological ordering. In the loop, a node X is selected that has no incoming edge in M. The *min* aggregate is necessary to select from multiple candidates. Later in the loop, all outgoing edges of X are removed from M. The output of the loop is the variable S which aggregates the selected values of X using the constant z and the function $f(S, X)$ (e.g. concatenation). If the graph is acyclic, the algorithm terminates after at most $|M|$ iterations; otherwise, it eventually fails to find any A in the statement 3 and, depending on the subtle details of the statement semantics, either causes an exception or loops indefinitely.

Modeling of a while-loop requires an extension to relational algebra and transitive closure is the most obvious candidate. Each iteration of the loop changes the state of the program – the variables M and S – thus, there is a relation B which models the loop-body behavior using tuples $\langle M, S, M', S' \rangle$. Together with the while-head condition H, the transitive closure $L = \sigma_{M'=\emptyset}(\sigma_{M\neq\emptyset}B)^*$ models the loop. Unfortunately, it requires nested relational algebra to represent the relation-valued attribute M; although nested relational algebra can be simulated with plain relational algebra [16], this is not the case with transitive closure. Thus, representing the example code in the algebraic world requires transitive closure over nested relational algebra, which includes expensive atomic operations like equality test over sets and it is difficult to reduce it to implementable physical operators.

A natural response to the problems of algebraic representation is switching to Datalog where the while loop may be handled easily using recursion. However, the Algorithm 1 demonstrates several obstacles in the Datalog approach.

3.1 Naïve approach

The following rule forms a naïve Datalog implementation of the loop body:

$$\begin{aligned}&\mathit{state}(M', S') \leftarrow \mathit{state}(M, S), \mathit{stmt}3(X, M),\\ &\quad \mathit{stmt}4(S', S, X), \mathit{stmt}5(M', M, X).\end{aligned}$$

The predicates *stmt*3, *stmt*4, and *stmt*5 implement the behavior of the statements 3, 4, and 5 of Algorithm 1. This approach creates extremely inefficient representation because any model of this program contains ground atoms for *stmt*3, *stmt*4, and *stmt*5 representing all satisfiable variable assignments for the statements regardless of its reachability during an execution. In addition, statement clauses (not shown here) violate safety rules as some of the variables are bound only by functional symbols. Consequently, this approach is not suitable for bottom-up evaluation in Datalog style.

3.2 Using function symbols for relational algebra

The following, improved representation may be derived from the previous one using the Magic-sets transformation [1]:

$$\begin{aligned}&\mathit{state}1(M) \leftarrow m0(M).\\ &\mathit{state}2(M, z) \leftarrow \mathit{state}1(M).\\ &\mathit{state}3(M, S) \leftarrow \mathit{state}2(M, S), M \neq \emptyset.\\ &\mathit{state}3(M, S) \leftarrow \mathit{state}6(M, S), M \neq \emptyset.\\ &\mathit{state}4(M, S, X) \leftarrow \mathit{state}3(M, S),\\ &\quad X = \pi_{min(A)}(\pi_A M \setminus \pi_B M).\\ &\mathit{state}5(M, S', X) \leftarrow \mathit{state}4(M, S, X), S' = f(S, X).\\ &\mathit{state}6(M', S) \leftarrow \mathit{state}5(M, S, X),\\ &\quad M' = M \setminus \sigma_{A=X} M.\\ &\mathit{state}7(S) \leftarrow \mathit{state}2(M, S), M = \emptyset.\\ &\mathit{state}7(S) \leftarrow \mathit{state}6(M, S), M = \emptyset.\end{aligned}$$

The predicate state_i indicate the reachability of a particular variable assignment in the beginning of statement i of Algorithm 1. The relational statements are represented as equality statements containing function symbols from relational algebra. The rules implement a state machine simulating the execution of the original program; the model expansion and safety problems mentioned above disappeared.

The relational statements are implemented using a non-Datalog mechanism; thus, this approach is far from being a suitable intermediate representation for mixed code.

3.3 Nesting and unnesting relational variables

In this section, the relational algebra was hoisted to Datalog level using the *unnest* predicate and the *nest* aggregate. This approach is equivalent to nested relational algebra. The predicate $unnest(M, A, B)$ performs the membership test $\langle A, B\rangle \in M$; the generalized aggregate $nest(A, B)$ collects all $\langle A, B\rangle$ pairs and combine them into a set (see [3] for exact definiton of semantics of aggregates in Datalog):

$state1(M) \leftarrow m0(M).$
$state2(M, z) \leftarrow state1(M).$
$cond2(M) \leftarrow state2(M, S), unnest(M, A, B).$
$state3(M, S) \leftarrow state2(M, S), cond2(M).$
$state3(M, S) \leftarrow state6(M, S), cond2(M).$
$cond3(M, B) \leftarrow state3(M, S), unnest(M, A, B).$
$state4(M, S, min(A)) \leftarrow state3(M, S),$
$\quad unnest(M, A, B), \neg cond3(M, A).$
$state5(M, S', X) \leftarrow state4(M, S, X), S' = f(S, X).$
$state6(nest(A, B), S) \leftarrow state5(M, S, X),$
$\quad unnest(M, A, B), A \neq X.$
$state7(S) \leftarrow state2(M, S), \neg cond2(M).$
$state7(S) \leftarrow state6(M, S), \neg cond2(M).$

This approach unifies the means used for procedural and relational fragments. Nevertheless, it suffers from the stratification required by the *nest* aggregate and the need to incorporate all live variables into single $state_i$ atom.

3.4 Numbering iterations

The following code illustrates the approach we finally used. The argument T representing time (more exactly, the number of iteration) was introduced to almost all rules. It allowed dissolution of the original $state_i$ predicates: $state_i(T)$ indicates reachability of the statement i at time T.

$m2(1, A, B) \leftarrow m0(A, B).$
$s2(1, z).$
$state2(1).$
$state2(T + 1) \leftarrow branch23(T).$
$cond2(T) \leftarrow state2(T), m2(T, A, B).$
$branch23(T) \leftarrow state2(T), cond2(T).$
$cond3(T, B) \leftarrow branch23(T), m2(T, A, B).$
$x4(T, min(A)) \leftarrow branch23(T),$
$\quad m2(T, A, B), \neg cond3(T, A).$
$s2(T + 1, f(S, X)) \leftarrow branch23(T),$
$\quad s2(T, S), x4(T, X).$
$m2(T + 1, A, B) \leftarrow branch23(T),$
$\quad m2(T, A, B), x4(T, X), A \neq X.$
$branch27(T) \leftarrow state2(T), \neg cond2(T).$
$return(S) \leftarrow branch27(T), s2(T, S).$

Variable values are represented independently: $x_i(T, V)$ determines the value V of the variable X before entering statement i at time T. In the case of relational-valued variable M, the relation is unnested and its tuples are represented by individual instances of the atom $m_i(T, A, B)$.

Relational (and in general, complex-valued) Datalog variables and terms are no longer needed – every term is an atomic value.

Note that the dissolution of $state_i$ predicates created an opportunity for optimization: Every atomic statement modifies only one variable; therefore, only one clause is required for each statement, specifying the new variable value. For a variable, it is sufficient to have the value specified only at reference points (for S and M, it is the beginning of the statement 2), provided that at least one reference point lies on any path from any definition to any usage of this variable. In our case, the value for the reference point 2 is specified twice because there are two control paths leading to this point.

The execution of rules implementing statements is guarded by trigger predicates $branch_{i,j}$ which signalize passing from the statement i to the statement j. These predicates are controlled by conditions and their negations; in our case, the predicate *cond*2.

3.5 Requirements on the logic language

In our example Algorithm 1, there is a loop-carried dependence from the variable M through X to the next M value which involves negation and aggregation. This is reflected in the presence of negation and aggregation in the mutual recursion between the predicates $x4$ and $s2$. Our representation is therefore unstratifiable; the unstratifiability is inherent to Algorithm 1 since the length of the chain of negations generated by the loop is unlimited. Consequently, no stratification may exist for any Datalog-like representation of this example.

This forces us to use the concept of *local stratification*. Note that in pure Datalog without function symbols, local stratification is almost equivalent to stratification [2]. However, our system does use function symbols to generate the time values T and to implement built-in operators and functions of the procedural language and of SQL.

3.6 Long-range variable passing

In our example Algorithm 1, each iteration of the loop modifies each variable. However, a loop body may contain conditional statements; thus, a variable may become unmodified in some iterations. In this case, there must be a mechanism to pass the unmmodified variable value through the loop body. A simple implementation of such mechanism is the following clause:

$$value_V(T+1,X) \leftarrow value_V(T,X), cond(T).$$

For every scalar variable V and for every iteration T, a ground atom $value_V(T,X)$ is present in the model indicating the value X of the variable. Consequently, the model is of the size $\Omega(\tau v)$ where τ is the execution time of the procedural program and v is the number of variables in the program. For non-scalar variables the cost is even larger, because the argument X (or more arguments) encodes an individual element of a relation or an array, therefore there are as many ground atoms as the size of the variable. Evaluating the complete model is thus unacceptably costly.

Fortunately, the cost of unmodified variable passing may be lowered to $O(\tau\ log(v))$. The principle is depicted in Fig. 3 – instead of copying the variable value on every iteration, there are several layers of preferred points in time and copying is performed at these preferred points. Preferred points of layer k occur at the distance of 2^k iterations and copying is allowed either to a point of higher preference or to a point of access to the variable. The thick lines in Fig. 3 show how a variable is passed when it is accessed in the three points marked at the time axis.

The number of copies done between two accesses is $O(log\Delta)$ where Δ is the time distance between the accesses, i.e. the length of the passing range. Since an atomic statement may access only a limited number of variables, the number n of passing ranges is proportional to the execution time, i.e. $n = c * \tau$. Since the ranges may not intersect for the same variable, their sum across of all variables is $\sum_{i=1}^{n} \Delta_i \leq \tau v$. Consequently, the total cost of copying is proportional to

$$\sum_{i=1}^{n} log(\Delta_i) \leq n\ log\frac{\sum_{i=1}^{n}\Delta_i}{n} \leq n\ log(\frac{\tau v}{n}) = \tau c\ log\frac{v}{c}$$

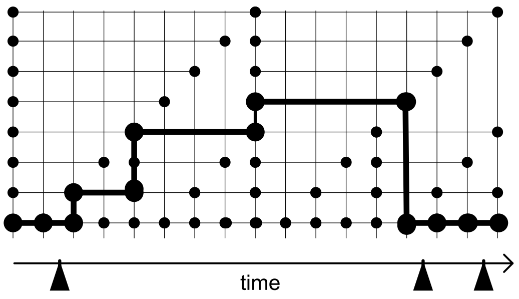

Fig. 3. Long-range variable passing

4 Conclusion

We have proposed a promising approach to mixed procedural and relational code whose key element is a novel intermediate representation based on logic programming. With respect to the assumed bottom-up evaluation strategy, the intermediate language falls to the Datalog family. Special care was taken for effective evaluation of the intermediate language – our results show only $O(log\ v)$ degradation with respect to the native procedural evaluation of a code with v variables.

In this paper, we presented the principles of the proposed language and its use for the representation of procedural code. For the sake of clarity, we omitted many technical details and tricks that were necessary to achieve the reported effectiveness of the representation.

Whether our approach is viable, it can be shown only by successful implementation of the whole processing chain from Fig. 2. The design of the intermediate representation was only a necessary prerequisite before attempting the implementation.

Our approach was motivated by the lessons learned from the implementation of a parallel SPARQL engine [4] for the Semantic Web. When using a relational repository, many Semantic Web algorithms are most easily expressed as simple procedural algorithms over relational queries.

Thus, using a combined relational/procedural intermediate language may save the tedious and error-prone work associated with reformulating such algorithms in either purely relational (if ever possible) or purely procedural way. In addition, the mixed representation offers new opportunities for optimization.

If successful, the new architecture may become important for areas where database access is tightly coupled with non-trivial computing, including the Semantic Web, computational linguistics or some areas of e-science.

References

1. C. Beeri, R. Ramakrishnan: *On the power of magic.* The Journal of Logic Programming, 10(3–4), 1991, 255–299.
2. H. A. Blair, V. W. Marek, J. S. Schlipf: *The expressiveness of locally stratified programs.* Annals of Mathematics and Artificial Intelligence, 15(2), 1995, 209–229.

3. M. P. Consens, A. O. Mendelzon: *Low-complexity aggregation in graphlog and datalog.* Theoretical Computer Science 116(1), 1993, 95–116.
4. Z. Falt, D. Bednarek, M. Cermak, F. Zavoral: *On parallel evaluation of sparql queries.* In DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications, 2012, 97–102.
5. S. C. Goldstein, F. Cruz: *Meld: A logical approach to distributed and parallel programming.* Technical report, DTIC Document, 2012.
6. S. Greco, L. Palopoli, E. Spadafora: *Extending datalog with arrays.* Data & Knowledge Engineering 17(1), 1995, 31–57.
7. R. Guravannavar, S. Sudarshan: *Rewriting procedures for batched bindings.* Proceedings of the VLDB Endowment 1(1), 2008, 1107–1123.
8. A. Guzzo, D. Sacca: *Semi-inflationary datalog: A declarative database language with procedural features.* Artificial Intelligence Communications 18(2), 2005, 79–92.
9. M. Gyssens, D. Van Gucht: *The powerset algebra as a result of adding programming constructs to the nested relational algebra.* ACM 17, 1988.
10. G. Lausen, B. Ludäscher, W. May: *On active deductive databases: The Statelog approach.* Transactions and Change in Logic Databases, 1998, 69–106.
11. Y. A. Liu, S. D. Stoller: *From datalog rules to efficient programs with time and space guarantees.* In Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming. ACM, 2003, 172–183.
12. C. Nomikos, P. Rondogiannis, M. Gergatsoulis: *Temporal stratification tests for linear and branching-time deductive databases.* Theoretical Computer Science 342(2), 2005, 382–415.
13. L. Palopoli: *Testing logic programs for local stratification.* Theoretical Computer Science 103(2), 1992, 205–234.
14. H. J. Schek, M. H. Scholl: *The relational model with relation-valued attributes.* Information Systems 11(2), 1986, 137–147.
15. Y. Smaragdakis, M. Bravenboer: *Using datalog for fast and easy program analysis.* In Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers, vol. 6702, pp. 245. Springer-Verlag New York Inc, 2011.
16. J. Van den Bussche: *Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions.* Theoretical Computer Science 254(1-2), 2001, 363–377.
17. E. Visser: *Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5.* In Rewriting Techniques and Applications, Springer, 2001, 357–361.