Deductive Databases and Logic Programming

Dietmar Seipel
University of Würzburg, Germany

Salvador Abreu
NOVA LINCS, University of Évora, Portugal

Abstract

Logic programming is an excellent platform for implementing *declarative* rule–based systems in artificial intelligence (AI). It is also well–suited for mediating between declarative tools such as relational or *deductive databases*, semi–structured databases (XML), and semantic web databases (based on RDF or OWL). Often, domain–specific languages are used in expert systems based on user–defined infix operators.

For the evaluation of expert rule bases, the declarative *bottom–up* computation of Datalog known from deductive databases can be complemented with Prolog's *top–down* evaluation. We will review some related work on logic programming and deductive databases and describe some tools with examples for declarative logic programming in Prolog and Datalog for information systems in AI that have been implemented within the declarative toolkit Declare.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Logic and databases

Keywords and phrases Deductive Databases, Knowledge Bases, Logic Programming

Digital Object Identifier 10.4230/OASIcs.SLATE.2026.13

Funding This work is supported in part by UID/04516/NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) with the financial support of FCT.IP.

1 Declarative Logic Programming

Declarative systems based on logic programming can combine declarative query languages such as SQL, SPARQL and *Datalog* with the processing of documents in established and widely used data formats, such as XML, RDF, and OWL. The logic programming language *Prolog* is an established and effective *mediator technology* between these hybrid data formats, since it allows for programming with meta–predicates and conveys expressiveness through nondeterminism built into the language. Also, Prolog is very well–suited for implementing *domain–specific languages* [6] using user–defined infix operators [23]. This feature promotes user–friendly applications, in which the language being coded in is embedded in the logic tools.

In what follows, we sketch the history of logic programming and databases, and then we briefly describe some declarative tools based on logic programming that we have implemented and that are mediated using Prolog.

1.1 Logic Programming and Databases

The field of logic and databases is an outgrowth of work in logic programming by Kowalski [10] and Lloyd [11] and in databases by Codd [3] and Ullman [26]. More powerful systems can be developed than by either approach alone, such as knowledge–based and nonmonotonic reasoning systems. Early work in relational databases emphasized implementation techniques and optimization methods using a *bottom–up* approach to query answering. This work led to efficient database systems of importance to commercial applications. Logic–based efforts

emphasized a *top–down* proof procedure and theoretical issues related to AI problems. The major contribution of Codd was to understand that the application of the relational calculus and the relational algebra to databases could yield a *declarative query language* that allows for *optimization techniques*. Those who approached databases through logic were influenced by work in theorem proving by J. Alan Robinson [19] who developed a uniform method to perform deduction in logic. The following development of the programming language Prolog by Colmerauer and his group [4] was very important for AI applications. Prolog is a top–down problem solver, and recursion is also permitted in logic programming while early relational systems did not permit recursion in queries or views. Early emphasis by the logic programming community was on syntax and semantics, only later commercially available systems could handle larger scale industrial problems.

In 1977, Gallaire, Nicolas and Minker organized a workshop on deductive databases [8]. Whereas logic programs permit function symbols in logical atoms, relational and deductive databases are generally function–free and deal almost exclusively with a finite set of constants in relations. The extended field of *disjunctive* logic programming and *disjunctive* deductive databases had its beginnings in the 80s [13, 12, 15]. The theoretical results extend the theory of logic programs as developed by Lloyd [11] and Apt [1]. The work in disjunctive logic programming has also been shown to be important for representing and implementing non-monotonic and abductive reasoning, and knowledge–based systems. There have been several *surveys* on deductive databases, starting with Gallaire, Minker and Nicolas [9]. Minker [14] provides historical material, results in complexity and early systems. Ramakrishnan and Ullman [18] focus on implementation techniques in Datalog, and they briefly describe a number of projects that have implemented systems. Minker, Seipel and Zaniolo [16] provide a history of the field of logic and databases from the late 1950s to the present.

1.2 Declare, DDBase, and Other Tools

The declarative logic programming system Declare with its deductive database component DDBase provides a logic programming language Datalog* that combines the top–down reasoning of Prolog with the bottom–up reasoning of Datalog: In this paper, we will describe

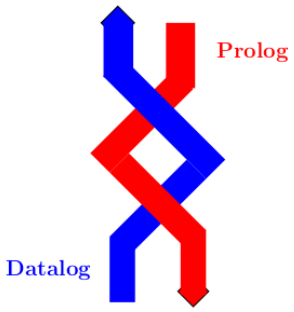

Figure 1 Prolog and Datalog for Declarative Logic Programming in DDBase.

the tools DDBase, FNQUERY, and DDQL of Declare with examples.

On the other hand, systems such as XSB Prolog [25] allows for top–down evaluation of logical rule systems in the programming style. XSB is an implementation of Prolog designed for knowledge–based systems, deductive databases, and AI applications in general. It extends Prolog with powerful features for efficient logical inference in situations not normally

handled by standard implementations. The key feature is *tabling (SLG resolution)*: it stores intermediate results, which can be used to detect and avoid infinite loops, thereby ensuring termination for many recursive queries. Tabling has been the object of implementation in other Prolog systems, including Ciao [7], YAP [5], and SWI [28].

In contrast, standard Prolog may loop indefinitely and recompute the same results multiple times. Tabling is useful for recursive queries, graph traversal, and reasoning over complex rule systems. These are common in deductive databases, AI, and knowledge representation. Typical application areas in deductive databases are rule–based querying and recursive data relationships; in artificial intelligence, logical inference and knowledge representation can benefit; in the semantic Web, ontologies and rule–based reasoning; in program analysis, static analysis and security verification. A typical example in tabled Prolog is the well–known *ancestor* rule set. In summary, tabling is a powerful extension to Prolog, geared for efficient logical inference, and especially useful for recursive and knowledge–based applications over relations that include graphs with cycles.

Compared to tabled Prolog, the database–style *bottom–up* evaluation of DDBase is easier to apprehend, as it separates normal Prolog from deductive database Datalog–style predicates explicitly. Moreover, Declare allows the embedding of Prolog calls into rule bodies, which are then evaluated as usual in logic programming languages. Declare can also make use of the more recently developed logic programming and non–monotonic reasoning (LPNMR) concept of *answer set programming* (ASP) on certain parts (strata) of the program and mediate between different types of hybrid knowledge bases.

The rest of this paper is organized as follows: Section 2 describes the tools DDBase, FnQuery, and DDQL of Declare in general, whereas Section 3 gives two more detailed examples for explaining the features of DDBase. Finally, Section 4 closes with a summary and some remarks on possible future extensions.

2 Tools in Declare

For complex knowledge bases, specialized problem solvers might be embedded in Declare, e.g. *answer set programming (ASP)* solvers. For many practical applications, however, the expressivity and performance of Declare is already sufficient.

In the system Declare, the *bottom–up computation* of Datalog known from deductive databases is complemented with Prolog's *top–down evaluation*. Datalog is embedded and interleaved with Prolog, and Prolog goals can even by called from Datalog in the sense of built–in predicates, thereby gaining expressiveness, see Figure 1.

In many application domains, the bottom–up point of view is more natural and easier to understand for the users. Other approaches from the literature seek to make Prolog more declarative, e.g. by forbidding certain predicates with side effects.

2.1 The Rule Knowledge Base DDBase

Modern knowledge base systems frequently need to combine a collection of databases in different formats: e.g., relational databases, Xml databases, rule bases, ontologies, etc. In the *deductive database system* DDBase [21], we can manage these different formats of knowledge and reason about them – even the file systems on different computers could be part of the knowledge base. The Graphical User Interface of DDBase is shown in Figure 2.

Often, it is necessary to handle different versions of a knowledge base; e.g., we might want to find out common parts or differences of two versions of a relational database.

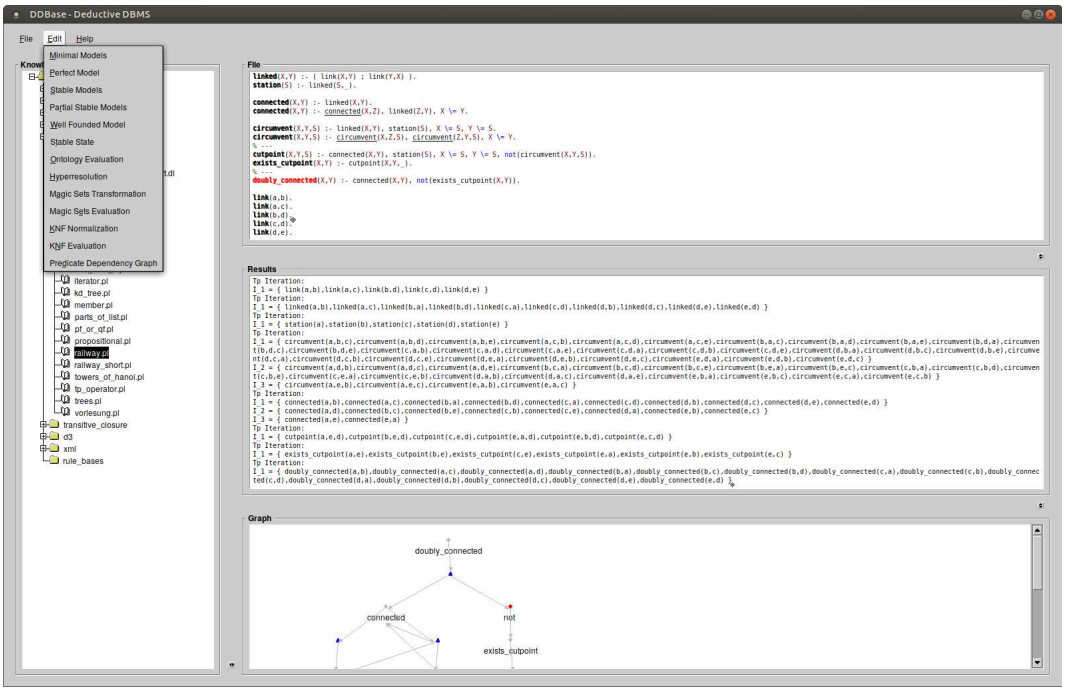

Figure 2 The Graphical User Interface of DDBase.

DDBase uses the domain–specific extension Datalog* of the standard query language Datalog known from deductive databases. Syntactically, Datalog* allows for general Prolog rules with function symbols and meta–predicates (one of them is default negation), which must be *stratified*, i.e. no recursion is allowed through meta–predicates. The user is responsible for avoiding infinite loops.

In contrast to standard Prolog rules, the rules are evaluated bottom–up after a static program analysis. DDBase uses abstractions of rule bases by predicate dependency and rule predicate graphs; also the proof trees of derived atoms can help to compare different versions of a rule base. A recent new contribution of DDBase also allows for a *semi–stratified* evalutaion of arbitrary default negation, that detects non–stratified parts by program analysis that are then evaluated using an ASP solver.

Moreover, it might be possible to have derivations joining rules with other formalisms of knowledge representation. Ontologies have shown their benefits in many applications of intelligent systems, and there have been many proposals for rule languages compatible with the semantic web stack, e.g., SWRL, the semantic web rule language. Recently, ontologies are used in hybrid systems for specifying the provenance of the different components.

DDBase can compute the *perfect model* for a Datalog* program by repeated iterations on the strata, see Figure 2.

In the next section, we will give two detailed motivating examples for DDBase, a longer one with stratified default negation on doubly connected networks, and a shorter one for grouping and counting the authors in a publication database with stratified aggregation.

2.2 XML Transformations, Queries, and Updates

With the tool FNQUERY [20], semi–structured (e.g. XML) data can be represented and queried nicely in a logic programming environment. It introduces a document object model for XML documents in Prolog, which is called *field notation*, and it simplifies dealing with

semi–structured data; instead of accessing a component of a Prolog fact by its argument position, one may do so non–positionally by its attribute name. One may also access nested components by a complex path or tree expression.

FnQuery implements a library FnPath of the domain–specific language XPath from Xml with predicates dealing with Xml data in field notation. Practical case studies have shown how to extract and transform information from Xml documents in a declarative style [20]. For some applications, the relevant information can be located automatically within the document, such that the extraction process becomes robust to certain changes of the document structure, over time.

As an example, consider an Xml representation of a tennis match of the following form:

```
<match>
 <set id="1">
  <game id="1" service="A">
   <point id="1" top="B" winner="A">
    <hit id="1" hand="forehand" time="00:00:42" x="0.17" y="-12.07"/>
    <hit id="2" hand="backhand" time="00:00:44" x="-0.49" y="5.89"/>
    <hit id="3" hand="backhand" time="00:00:46" x="-3.92" y="-3.42"/>
    <hit id="4" hand="forehand" time="00:00:48" x="3.56" y="2.06"/>
   </point>
   <point id="2" ...> ... </point> ...
  </game> ...
 </set> ...
</match>
```

The following DdBase query computes a list of tuples `[S, G, P, L, D]`, where for each set `S`, game `G`, and point `P`, it computes `L` as the average of the points lengths and `D` as the average of the points distances; it uses path expressions in FnPath to access the Xml document `Match`:

```
average_point_distances(Match, Tuples) :-
   ddbase_aggregate( [S, G, P, avg(L), avg(D)],
      ( Point := Match/set::[@id=S]/
           game::[@id=G, @service=Service]/point,
        P := Match/player::[@id=Service]@name,
        point_to_length_and_distance(Point, L, D) ),
      Tuples ).
```

2.3 The Deductive Database Query Language DDQL

DdQl [24] proposes a natural, keyword–based query interface for knowledge bases – including relational or deductive databases – based on contextual background knowledge such as suitable *join conditions* or *synonyms*. Join conditions could be extracted from existing referential integrity (foreign key) constraints of the database schema; they could also be learned from other, previous database queries, if the database schema does not contain foreign key constraints.

Given a textual representation – a word list – of a query to a relational database, one may parse the list into a structured term. The intelligent and cooperative part of the approach is to hypothesize the semantics of the word list and to find suitable links between the concepts mentioned in the query using contextual knowledge, more precisely join conditions between the database tables.

Figure 3 The Graphical User Interface of DdQl.

DdQl uses a knowledge–based parser based on an extension of *definite clause grammars* (DCG) that are interleaved with calls to the database schema to suitably annotate the tokens as table names, table attributes, attribute values or relationships linking tables. DdQl yields the possible queries in a special domain–specific rule language that extends Datalog, from which the user can choose one.

The deductive database query language DdQl has been described in detail with examples in another paper at SLATE 2023, cf. [22]. The example in Figure 3 shows the graphical user interface of DdQl for a relational database query asking for the list of salaries of department. This stimulates an aggregation query to group the salaries of the departments by listing all salaries per department. Since the `list` aggregation function list is not allowed in Sql (the typical aggregation funtions would be `sum`, `agg`, `min`, and `max`), this query can only be answered in DdBase.

3 DDBase Examples

Deductive databases offer recursive rules, which can be seen as an extension of Sql views, where only limited forms of recursion are possible. In DdBase, also stratified aggregation operations and default negation are possible; stratification means that the goal of the meta–operation (e.g. aggregation or default negation) has to be evaluated before the operation itself and cannot recursively depend on the meta–operation.

In deductive databases, we split a logic program P into subsets of rules (*strata*), such that a default negation in a stratum only refers to the same stratum or to lower strata. Then, the logic program can be evaluated stratum by stratum – beginning with the lowest stratum. This derives the so–called *perfect model* of the program. Such a partitioning into strata does not always exist, if there is default negation interleaved with recursion. Then, we need more elaborated evaluation methods and semantics such as *well–founded* or *stable models*.

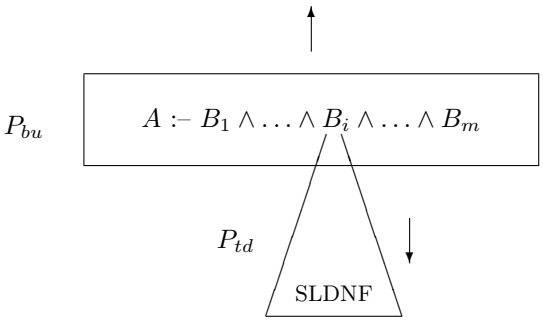

Figure 4 Forward Chaining with Prolog Built–Ins.

DDBASE allows for rules in Prolog syntax, which are – however – evaluated bottom–up. The program is split into a of set P_{bu} of rules that are evaluated bottom–up and another set P_{td} of rules that are evaluated top–down. The body atoms B_i of a rule in P_{bu} are evaluated top–down with Prolog's SLDNF resolution. If the body evaluates to true, then the head A of a rule is derived.

The consequence operator of DDBASE for forward chaining with Prolog built–ins evaluates a Prolog program P_{bu}, which is given as a list of rules and is not part of the Prolog database. It uses a helper Prolog program P_{td} that is stored separately.

3.1 Doubly Connected Network with Stratified Default Negation

The following recursive DATALOG$^{\text{not}}$ program of [2] decides whether a network is doubly connected: $P_{dc} = \{ r_1, \ldots, r_{10} \}$. For visualization, we use the subway network (Schnellbahn Netzplan) of Munich, cf. Figure 5. We can solve it and explain it more declaratively with the combination of bottom–up and top–down of DDBASE, cf. Figure 4, rather than it could be done in Prolog or XSB alone; in realistic extensions, there could be additional body atoms B_i to be solved in Prolog during a stratified bottom–up evaluation.

The first 5 rules compute stations and connected nodes X and Y from a base relation *link*, which is first made symmetric in the form of *linked* and for which we also define its transitive closure, expressed as *connected*, as follows:

$$\begin{aligned} r_1 &= \mathit{linked}(X,Y) \text{ :– } \mathit{link}(X,Y), \\ r_2 &= \mathit{linked}(X,Y) \text{ :– } \mathit{link}(Y,X), \\ r_3 &= \mathit{station}(X) \text{ :– } \mathit{linked}(X,Y), \\ \\ r_4 &= \mathit{connected}(X,Y) \text{ :– } \mathit{linked}(X,Y), \\ r_5 &= \mathit{connected}(X,Y) \text{ :– } \mathit{linked}(X,Z) \wedge \mathit{connected}(Z,Y) \wedge X \neq Y. \end{aligned}$$

The following rules compute triples X, Y, S, such that there exists a connection from X to Y that circumvents S (rules r_6 and r_7):

$$\begin{aligned} r_6 &= \mathit{circumvent}(X,Y,S) \text{ :–} \\ &\qquad \mathit{linked}(X,Y) \wedge \mathit{station}(S) \wedge X \neq S \wedge Y \neq S, \\ r_7 &= \mathit{circumvent}(X,Y,S) \text{ :–} \\ &\qquad \mathit{circumvent}(X,Z,S) \wedge \mathit{circumvent}(Z,Y,S) \wedge X \neq Y. \end{aligned}$$

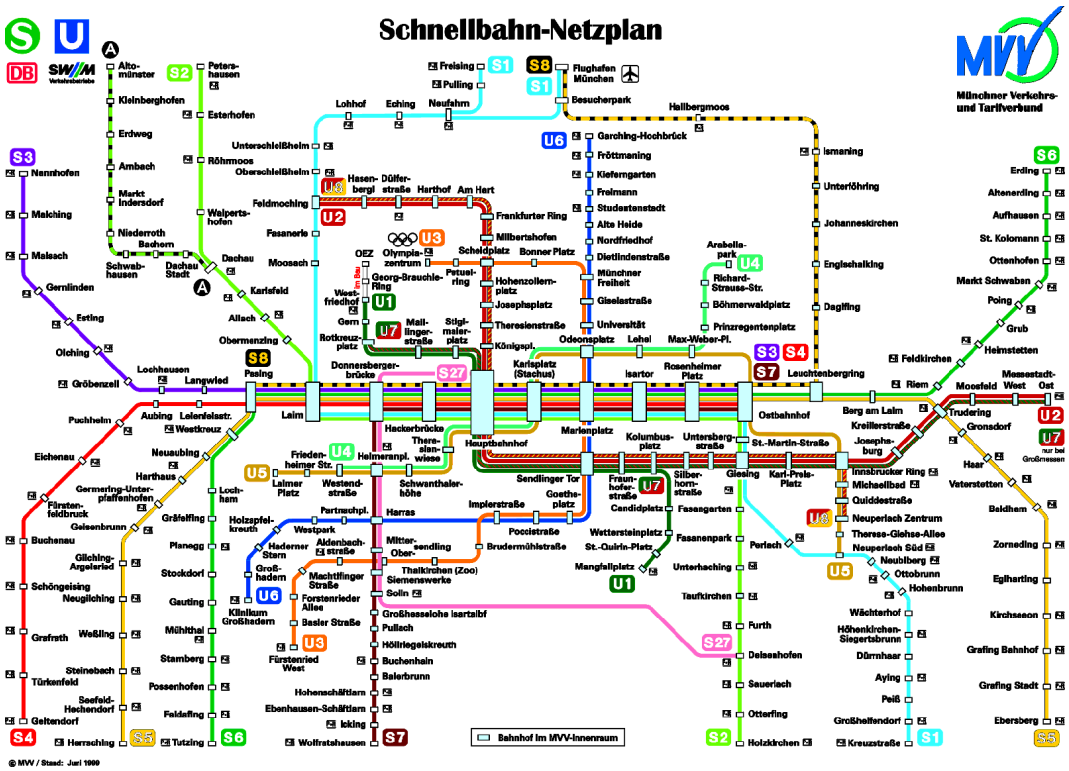

Figure 5 Subway Network of Munich.

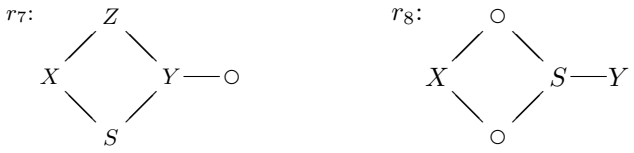

Figure 6 Circumvent and Cutpoint.

S is a cutpoint between X and Y, if X and Y are connected and S cannot be circumvented on this connection (rule r_8), see Figure 6:

$$\begin{aligned} r_8 \;=\; & \mathit{cutpoint}(X,Y,S) \text{ :–} \\ & \mathit{connected}(X,Y) \wedge \mathit{station}(S) \wedge \\ & X \neq S \wedge Y \neq S \wedge \\ & \textsf{not } \mathit{circumvent}(X,Y,S). \end{aligned}$$

According to rule r_8, a station S is a cutpoint, if X and Y are connected, S is different from X and Y (no matter where S occurs on a path from X to Y), and S cannot be circumvented. In Figure 6, on the left, S can be circumvented, since there exists another path through Z; on the right, S cannot be circumvented, since the only path from X to Y has to pass S.

Two nodes X and Y are doubly connected, if they are connected and there exists not cutpoint between X and Y (rules r_9 and r_{10}):

$$\begin{aligned} r_9 \;=\; & \mathit{exists\_cutpoint}(X,Y) \text{ :–} \\ & \mathit{cutpoint}(X,Y,S), \\ r_{10} \;=\; & \mathit{doubly\_connected}(X,Y) \text{ :–} \\ & \mathit{connected}(X,Y) \wedge X \neq Y \wedge \\ & \textsf{not } \mathit{exists\_cutpoint}(X,Y). \end{aligned}$$

In summary, the DATALOG$^{\text{not}}$ program $P_{dc} = \{ r_1 \ldots, r_{10} \}$ has, e.g., the following *stratification*

$$\begin{aligned} \Pi_1 \;&=\; \{\, \mathit{link}, \mathit{linked}, \mathit{station}, \mathit{connected}, \mathit{circumvent} \,\}, \\ \Pi_2 \;&=\; \{\, \mathit{cutpoint}, \mathit{exists\_cutpoint} \,\}, \\ \Pi_3 \;&=\; \{\, \mathit{doubly\_connected} \,\}, \end{aligned}$$

with the corresponding strata $P_1 = \{r_1, \ldots, r_7\}$, $P_2 = \{r_8, r_9\}$, $P_3 = \{r_{10}\}$. The *dependency graph* $G^d_{P_{dc}}$ is shown in Figure 7. Besides the marked stratification, there exist others.

In DDBASE, this program P_{dc} can be evaluated using a stratified bottom–up fixpoint computation. Many intermediate facts are computed during the evaluation, but this poses no problem to the DDBASE system. In practice, the facts for `link/2` and `station/1` were derived by the following Prolog helper rules

```
link(X, Y) :-
   line(_, Stations), append(_, [X, Y|_], Stations).
station(S) :-
   line(_, Stations), member(S, Stations).
```

and facts for `line/2` of the form

```
line(s1, [
   freising, moosach, laim, donnersbergerbruecke, ostbahnhof,
   giesing, neuperlachsued, kreuzstrasse ]).
...
```

Together with P_{dc}, these rules form a Datalog* program P_{bu}. The helper program is formed by the standard Prolog rules for `append/3` and `member/2`. We got good performance on stratified evaluation, taking less than 17 seconds as more than 60.000 facts are generated and processed, running on a single–threaded Intel Core i7–8565U CPU.

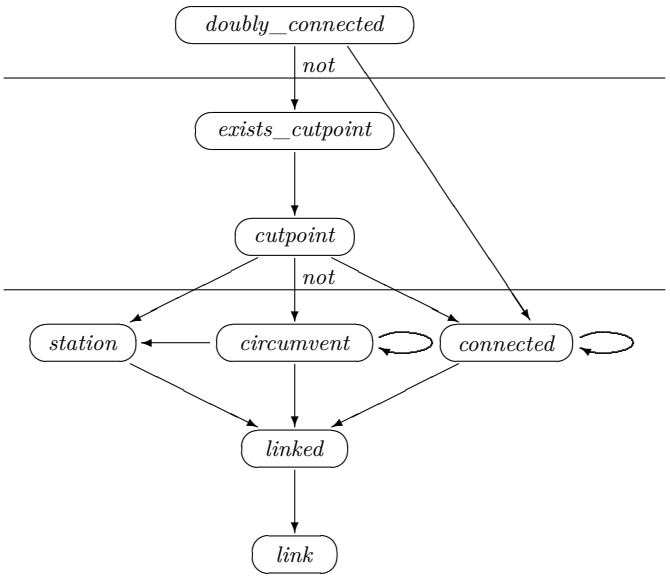

Figure 7 Dependency Graph with Stratification in 3 Strata.

3.2 Count Authors with Aggregation

Given a base predicate about publications, the following DDBASE code can compute coauthor counts and lists:

```
co_author(Author_1, Author_2) :-
   paper(Author_1, Paper),
   paper(Author_2, Paper),
   Author_1 \= Author_2.

co_authors(Author, Count, Co_Authors) :-
   [Author, Count, Co_Authors] <==
       ddbase_aggregate( [A, count(B), list(B)],
          co_author(A, B) ).
```

A call `Xs <== ddbase_aggregate(Template, Goal)` computes all answers `Template` to the goal `Goal`, and successively unifies the list `Xs` with these answers. Thus, lists can be generated by this construction, which is not possible by aggregation in SQL, and the aggregation functions can be self–defined.

For an example database on publications with binary facts about `paper`, transitive coauthors can already be computed in standard deductive databases. In DDBASE, the coauthor distance, Erdős number or cliques can be computed by built–in calls to Prolog or other external languages. If the base facts are given in domain–specific terms for XML, then FNQUERY can handle the transformation from this XML to Datalog.

In DDBASE, embedded calls to answer set programming (ASP solvers) can be used to evaluate non–stratified default negation – which can be detected by program analysis. In DDQL, natural queries in the form of Google–like buzzword lists can be evaluated. We are presently working on an extension with automated speech recognition (ASR) and contextual logic programming.

4 Conclusions and Future Directions

The deductive database component DDBASE of the logic programming toolkit Declare helps to effectively build applications of intelligent information systems in a declarative way. Datalog* can embed Prolog calls into the standard bottom–up evaluation of Datalog. Like in embedded SQL, Datalog can now be embedded into the programming language Prolog, and Prolog built–ins can be embedded into Datalog.

Relational databases and XML documents can be accessed from Datalog* by ODBC and FNQUERY, respectively. We have not shown ODBC here, since it is a standard feature of programming languages such as Prolog. The query language DDQL is another tool of Declare, which has been described in detail in another paper.

It is also our belief that an integration of the program–structuring abilities brought by contexts, especially along the lines of temporal contextual logic programming [17], can be leveraged to effectively and naturally build further *mediator programming* features, in which multiple versions of ontologies may coexist and be queried in a unified way. We are working on an integration of TCXLP into DDBASE and thereby provide a scalable and efficient means for keeping control over large–scale web or database applications. Such a system would inherit the underlying components' properties of verifiability and provability.

References

1 Krzysztof R. Apt. Logic Programming. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 493–574. Elsevier and MIT Press, 1990. doi:10.1016/b978-0-444-88074-1.50015-9.

2 Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Surveys in computer science. Springer, 1990. URL: https://www.worldcat.org/oclc/20595273.

3 E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970. doi:10.1145/362384.362685.

4 Alain Colmerauer, Henry Kanoui, R. Pasero, and P. Roussel. Un systeme de communication homme–machine en francais. Technical report, Groupe de Intelligence Artificielle Universitae de Aix–Marseille II, 1973.

5 Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *Theory Pract. Log. Program.*, 12(1-2):5–34, 2012. doi:10.1017/S1471068411000512.

6 M. Fowler. *Domain–Specific Languages*. Addison–Wesley, 2011.

7 Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. An Overview of Ciao and its Design Philosophy. *Theory Pract. Log. Program.*, 12(1-2):219–252, 2012. doi:10.1017/S1471068411000457.

8 Herve Gallaire and Jack Minker, editor. *Logic and Data Bases*. Plenum Press, 1978.

9 Hervé Gallaire and Jack Minker and Jean–Marie Nicolas. Logic and Databases: A Deductive Approach. *ACM Comput. Surv.*, 16(2):153–185, 1984. doi:10.1145/356924.356929.

10 Robert A. Kowalski. Predicate Logic as Programming Language. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 569–574. North-Holland, 1974.

11 John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987. doi:10.1007/978-3-642-83189-8.

12 Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. Logic Programming. MIT Press, 1992.

13 Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988. doi:10.1016/c2013-0-07699-2.

14 Jack Minker. Logic and Databases: A 20 Year Retrospective. In Dino Pedreschi and Carlo Zaniolo, editors, *Logic in Databases, International Workshop LID'96, San Miniato, Italy, July 1-2, 1996, Proceedings*, volume 1154 of *Lecture Notes in Computer Science*, pages 3–57. Springer, 1996. doi:10.1007/BFb0031734.

15 Jack Minker and Dietmar Seipel. Disjunctive Logic Programming: A Survey and Assessment. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *Lecture Notes in Computer Science*, pages 472–511. Springer, 2002. doi:10.1007/3-540-45628-7_18.

16 Jack Minker, Dietmar Seipel, and Carlo Zaniolo. Logic and Databases: A History of Deductive Databases. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 571–627. Elsevier, 2014. doi:10.1016/B978-0-444-51624-4.50013-7.

17 Vítor Nogueira and Salvador Abreu. Temporal Contextual Logic Programming. *Electron. Notes Theor. Comput. Sci.*, 177:219–233, 2007. doi:10.1016/j.entcs.2007.01.025.

18 Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Deductive Database Systems. *J. Log. Program.*, 23(2):125–149, 1995. doi:10.1016/0743-1066(94)00039-9.

19 John Alan Robinson. A Machine–Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965. doi:10.1145/321250.321253.

20 Dietmar Seipel. Processing XML-Documents in Prolog. In *Proc. 17th Workshop on Logic Programming (WLP 2002)*, 2002.

21 Dietmar Seipel. Knowledge Engineering for Hybrid Deductive Databases. In *Proceedings 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming, WLP 2015 / WLP 2016 / WFLP 2016*, volume 234 of *EPTCS*, pages 1–12, 2017. doi:10.4204/EPTCS.234.1.

22 Dietmar Seipel, Benjamin Förster, Magnus Liebl, Marcel Waleska, and Salvador Abreu. Querying Relational Databases with Speech–Recognition Driven by Contextual Knowledge. In Alberto Simões, Mario Marcelo Berón, and Filipe Portela, editors, *12th Symposium on Languages, Applications and Technologies, SLATE 2023, Vila do Conde, Portugal, June 26-28, 2023*, OASIcs, pages 6:1–6:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/OASIcs.SLATE.2023.6.

23 Dietmar Seipel, Falco Nogatz, and Salvador Abreu. Domain–specific Languages in Prolog for Declarative Expert Knowledge in Rules and Ontologies. *Comput. Lang. Syst. Struct.*, 51:102–117, 2018. doi:10.1016/j.cl.2017.06.006.

24 Dietmar Seipel, Daniel Weidner, and Salvador Abreu. Intelligent Query Answering with Contextual Knowledge for Relational Databases. In *10th Symposium on Languages, Applications and Technologies, SLATE*, volume 94 of *OASIcs*, pages 16:1–16:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/OASIcs.SLATE.2021.16.

25 Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *CoRR*, abs/1012.5123, 2010. arXiv:1012.5123.

26 Jeffrey D. Ullman. *Principles of Database and Knowledge–Base Systems, Volume II*. Computer Science Press, 1989.

27 Jeffrey D. Ullman. The Theory of Deductive Database Systems. In *Intellectual Leverage: Thirty–Fifth IEEE Computer Society International Conference, Compcon Spring '90, San Francisco, California, USA, February 26 – March 2, 1992, Digest of Papers*, pages 496–502. IEEE Computer Society, 1990. doi:10.1109/CMPCON.1990.63730.

28 Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory Pract. Log. Program.*, 12(1-2):67–96, 2012. doi:10.1017/S1471068411000494.