Modern Datalog: Concepts, Methods, Applications

Markus Krötzsch
Knowledge-Based Systems Group, TU Dresden, Germany

Abstract

Pure Datalog is arguably the most fundamental rule language, elegant and simple, but also often too limited to be useful in practice. This has motivated the introduction of many new expressive features, ranging from datatypes and related functions, over aggregates and semi-ring generalisations, to existential quantifiers and complex terms. In spite of their variety, all these approaches remain true to the nature of Datalog as a direct, pattern-based way of computing on structured data. We therefore find that a modern notion of Datalog is emerging, distinctly different from other approaches of logic programming and with its own set of related methods and applications.

In this course, we introduce Datalog and its most common extensions, and explain when and how these features can be used together (which is often, but not always, safe to do). We further look at modern Datalog systems and some of their primary use cases. Hands-on work with Datalog and its extensions is done with the free Datalog engine Nemo. The course is accessible to all audiences and does not assume specific prior knowledge.

2012 ACM Subject Classification Computing methodologies → Logic programming and answer set programming; Theory of computation → Constraint and logic programming; Theory of computation → Database query languages (principles); Theory of computation → Logic and databases

Keywords and phrases Datalog, query language, knowlegde representation and reasoning, logic programming, Horn logic, SPARQL, datatypes and aggregation, lecture notes, tutorial

Digital Object Identifier 10.4230/OASIcs.RW.2024/2025.1

Category Invited Paper

Supplementary Material *Software (Source Code, reasoner)*: `https://github.com/knowsys/nemo`
Software (Source Code, Web IDE): `https://github.com/knowsys/nemo-web/`
Software (Source Code, proof tree visualiser): `https://github.com/imldresden/nev`

Funding This work is partly supported by DFG in project 389792660 (TRR 248, Center for Perspicuous Systems), by BMFTR in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), and by BMFTR and DAAD (German Academic Exchange Service) in project 57616814 (SECAI, School of Embedded and Composite AI).

Acknowledgements The practical hands-on exercises in these lecture notes rely on the Nemo rule engine. Thanks are due to all past and present contributors, but especially to Lukas Gerlach, Alex Ivliev, and Maximilian Marx for getting key features to work in time for this lecture. The proof tree visualisation in Figure 3 has been implemented by Tobias Wieland with contributions from Julián Méndez and Lukas Gerlach.

1 Introduction

The one feature that immediately stands out when introducing Datalog is its simplicity. Anyone who understands the basic idea of encoding information in relational facts, such as $\mathsf{hasFather}(\mathsf{alice}, \mathsf{bob})$, will have little difficulty in interpreting rules such as

$$\mathsf{hasUncle}(x, z) \leftarrow \mathsf{hasFather}(x, y), \mathsf{hasBrother}(y, z), \tag{1}$$

at least after being told that variables x, y, z are placeholders for arbitrary elements, and that $\leftarrow$ means *if* and the comma means *and*. Datalog *programs* are essentially collections

```
1           hasUncle(X, Z) :- hasFather(X, Y), hasBrother(Y, Z) .
2           hasUncle(x, z) :- hasFather(x, y), hasBrother(y, z) .
3         hasUncle(?x, ?z) :- hasFather(?x, ?y), hasBrother(?y, ?z) .
4      [?x, :hasUncle, ?z] :- [?x, :hasFather, ?y], [?y, :hasBrother, ?z] .
5  hasUncle(x, z) distinct :- hasFather(x, y), hasBrother(y, z) ;
6       [(has-uncle ?x ?z)    [?x :has-father ?y] [?y :hasBrother ?z]]
7 hasUncle(p: x, uncle: z) :- hasFather(child: x, father: y), hasBrother(p: y, brother: z) .
8 { ?X :hasFather ?Y . ?Y :hasBrother ?Z .} => { ?X :hasUncle ?Z } .
```

Figure 1 The Datalog rule of equation (1) as written in various common systems that support Datalog: Line 1 *Prolog* [34], which agrees with *ASP-Core-2* [12] here, L2 *Soufflè* [36], L3 *Nemo* [35], L4 *RDFox* [56], L5 *Logica* [46], L6 *Datomic* [17], L7 *Percival* [73], L8 *N3* rule language [68]

of such rules, intended to be applied exhaustively to a given set of facts.[1] Entailments produced by applying one rule may potentially allow us to apply other rules in new ways. A trademark feature of Datalog then is that programs may be *recursive* in the sense that a rule's application may (indirectly) lead to entailments that require us to apply the same rule again. Nevertheless, it does not matter how we chose to apply rules: after finitely many steps, we will always arrive at a unique set of entailed facts – the output of the Datalog program.

Given this simple framework, it is astonishing that Datalog has been applied to solve many complex tasks, such as database query answering, source code analysis, and ontological reasoning, which typically require much more complicated programming and query languages. Versatility and simplicity together may explain why Datalog continues to be widely used and implemented even decades after its inception, with new systems coming up regularly.

Nevertheless, aspiring users of Datalog are often faced with considerable difficulties, since the landscape of Datalog languages and systems is fractured. Even basic syntax is all but uniform. Figure 1 illustrates how rule (1) would be expressed in various standards and tools.[2] The traditional syntax in L1 was coined over 40 years ago for Prolog, but more recent systems have sought alignment with standards like SQL, SPARQL, or JSON that support Unicode, IRIs, and specific data structures. Moreover, Datalog is often extended with further features, such as negation, aggregation, or datatype-related functions, which tend to have even more varied syntax. Thankfully, there is still a broad agreement on the semantics of basic rules, with only very few exceptions. Nevertheless, there are important differences in terms of computational abilities and interfaces that modern tools provide, since no system is built to address all relevant uses of Datalog, so practitioners have to choose carefully.

The goal of these lecture notes is to offer an overview of *contemporary Datalog* – the evolved version(s) of Datalog as used in today's applications – with its main features and usage scenarios, and some of the underlying concepts and mechanisms. Our preferred perspective will be that of a researcher or practitioner who considers using Datalog for their own purposes, as opposed to the perspective of a developer of Datalog reasoners. We therefore aim to convey an understanding of the logical language and the technical capabilities of modern systems, and of the requirements and considerations that have led to their current state. Readers may thus find guidance for their own work with Datalog and its extensions, and,

[1] Implementations may not always literally "apply rules exhaustively" to produce their outputs, but the idea can always serve as an intuitive view of the semantics of Datalog and many of its extensions.

[2] Figure 1 omits the excessively verbose XML syntax of the W3C standard RIF [39]; RIF's more readable *Presentation Syntax* supports rules as in L3.

ideally, some inspiration for new research and applications

Guide to the Reader

The text aims to be fully accessible with minimal prior knowledge, but also covers some more specialised topics that can be of interest to advanced readers who may skip the introductory parts. Implementation methods will largely be omitted,[3] whereas motivations, formal foundations, practical remarks, and pointers to further research will be included. The text is organised in such a way that many parts can be skipped. Whenever earlier content is relevant to the reader's understanding, pointers will be given.

▶ Note 1 (Use of Notes). Notes within the text are used to discuss specific aspects or alternative approaches that are not essential to the understanding of the main text. They can safely be skipped.

Besides smaller examples throughout, the text contains three extended *hands-ons* in Sections 2, 5, and 9. These sections invite readers to try out their insights on worked examples. A no-install testing environment is provided (see Section 2). Readers with more foundational interests can completely skip these sections without loss of continuity.

The outline of the text is as follows:

- Section 2: First hands-on, getting to know our accompanying try-out system
- Section 3: Extended discussion on why and how Datalog is used at all, including many examples of previously reported use cases
- Section 4: An introduction to the mathematical syntax and semantics of pure Datalog
- Section 5: Second hands-on, explaining how we can use pure Datalog with data from external sources
- Section 6: Negation in Datalog, and the related concept of stratification
- Section 7: Approaches of making datatypes and related functions available in Datalog
- Section 8: Aggregation functions in Datalog, building on concepts from Sections 6 and 7
- Section 9: Third hands-on, presenting a program that integrates and analyses real data
- Section 10: Overview of approaches that generalise Datalog through alternative (non-boolean) truth values

2 Hands-On: Trying Out Datalog Online

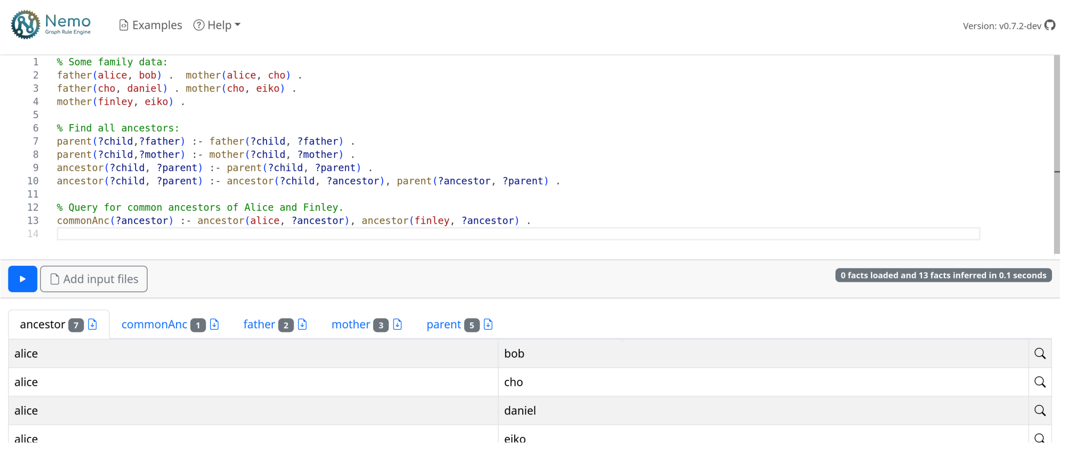
Throughout this text, we will look at some worked examples that illustrate how the theory can be put into practice. For this purpose, we will use the Web app of the rule reasoner Nemo [35], which can be accessed online at `https://tools.iccl.inf.tu-dresden.de/nemo/`. The Web application can execute complex Datalog programs in the browser and display the results. Alternatively, Nemo releases can also be downloaded and run locally as a command-line tool. Nemo's user manual can be found at `https://knowsys.github.io/nemo-doc/`.

For a first example, consider the program about simple family relations in Listing 1. A short link to this online example is `https://tud.link/ctdxps`, but it is also possible to simply copy the program manually. The Web application consists of a rich Datalog editor field and several controls, as shown in Fig. 2. Most important is the "play" button in the

[3] Classical abstract techniques like *semi-naive evaluation* and *magic sets* are widely described in lectures and textbooks (e.g., [2]), and many research papers have also put their focus on implementation aspects. The literature seems to offer more advice to those who want to program their own Datalog engines than to those who want to use them.

```
 9  % Some family data:
10  father(alice, bob) .  mother(alice, cho) .
11  father(cho, daniel) . mother(cho, eiko) .
12  mother(finley, eiko) .
13
14  % Find all ancestors:
15  parent(?child,?father) :- father(?child, ?father) .
16  parent(?child,?mother) :- mother(?child, ?mother) .
17  ancestor(?child, ?parent) :- parent(?child, ?parent) .
18  ancestor(?child, ?parent) :- ancestor(?child, ?ancestor), parent(?ancestor, ?parent) .
19
20  % Query for common ancestors of Alice and Finley.
21  commonAnc(?ancestor) :- ancestor(alice, ?ancestor), ancestor(finley, ?ancestor) .
```

Listing 1 Introductory example of a Datalog program



```
% Some family data:
father(alice, bob) .  mother(alice, cho) .
father(cho, daniel) . mother(cho, eiko) .
mother(finley, eiko) .

% Find all ancestors:
parent(?child,?father) :- father(?child, ?father) .
parent(?child,?mother) :- mother(?child, ?mother) .
ancestor(?child, ?parent) :- parent(?child, ?parent) .
ancestor(?child, ?parent) :- ancestor(?child, ?ancestor), parent(?ancestor, ?parent) .

% Query for common ancestors of Alice and Finley.
commonAnc(?ancestor) :- ancestor(alice, ?ancestor), ancestor(finley, ?ancestor) .
```

0 facts loaded and 13 facts inferred in 0.1 seconds

alice	bob
alice	cho
alice	daniel
alice	eiko

Figure 2 Rule reasoning in the browser with Nemo Web

lower left, which starts the reasoner. Once reasoning is finished, results are displayed below. By default, all derived relations are displayed. All reasoning is performed locally, without any data being sent to a server.

The workings of the example program should not be hard to follow, even without having introduced Datalog formally yet. The initial facts (L10–L12) provide some basic information on simple entities. Rules L15 and L16 derive facts about parenthood, and rules L17 and L18 define `ancestor` as the transitive closure of `parent`. Finally, L21 is asks for common ancestors of Alice and Finley. Lines with % are comments.

As a small exercise, readers are invited to try to add rules to define relations `sibling` and `auncle` (the binary relation between a niece or nephew and their aunt or uncle). Inequality `!=` can be used to prevent anybody from becoming their own sibling. If correct, `auncle(alice,finley)` should be derived.

▶ Note 2 (To the Future Reader). Reproducibility is always difficult, especially in the future. All online exercises in this chapter were fully functional at the time of its writing, but this may change: systems evolve, web sites vanish, bits rot. Future readers who find that some hyperlinks fail, yet wish to experience the original 2025 technology, have several options:

- Manually use current Nemo Web IDE. If `https://tools.iccl.inf.tu-dresden.de/nemo/` still exists, programs can likely be run there, possibly with some smaller syntactic modifications.

- Run original Nemo Web IDE locally. The web IDE used here is Nemo Web v0.9 and can be run locally with most browsers; see `https://github.com/knowsys/nemo-web/releases` for downloads.
- Install Nemo locally. All exercises are based on Nemo v0.9, which can also be run locally without the Web IDE; see `https://github.com/knowsys/nemo/releases` for downloads.

3 Datalog: What, Why, and How

Before turning towards concrete technical questions, we should take a moment to ask ourselves what benefits we expect from a logic-based language like Datalog, and what grounds we have for deeming it superior – at least in some aspects – over obvious alternatives. Surely, whatever we are up to could also be realised by a suitable program in a modern imperative language like Python or Rust, and possibly, if our needs are more modest, by a database query language like SPARQL or SQL. In this section, we will therefore look into (hoped for) benefits and potential application areas where matching requirements arise.

3.1 Why Datalog? Why Logic?

A distinguishing feature of Datalog is its clean, mathematically defined semantics. As we will see in Section 4, this semantics can equivalently be defined in several ways, each of which is elegant and well-motivated in its own right. Mathematical purists will find this enough to prefer such languages over more messy alternatives,[4] but we may wish to investigate further whether this also translates into concrete practical benefits. What follows are some of the main claims that are associated with logic-based computational paradigms in general, and with Datalog in particular.

Declarativity

Since its inception, logic programming was argued to be *declarative* in the sense that programmers rather describe the problem at hand than a step-by-step process for solving it. Without direct procedural control or low-level memory management, pure logic just specifies "what should be true", leaving the task of computation to the reasoner. However, logic programming is not free from operational concerns, and practical programs are often engineered to achieve efficient computation. Conversely, declarativity is also found in many other formalisms, including query languages, data processing languages, markup languages, and high-level programming languages. Rather than a rigid property of a formal language, declarativity therefore is an ideal to strive for in the design of languages *and* their concrete use in individual cases. It is often implied that this will enable further benefits down the road, and indeed declarativity may contribute to all of the following benefits.

System and Platform Independence

In practical fields of computation, declarativity is often considered synonymous with the ability to "run" the same specification on various systems and hardware platforms. An SQL

[4] For example, the SPARQL 1.1 standard includes half-textual descriptions of example algorithms to convey how the correct result of queries is to be determined in some cases [59], which is hard to analyse and has contributed to some undesired behaviour [57].

query should yield the same result on different database management systems, a PyTorch tensor operation may equivalently be executed on a CPU or on a GPU, and a Datalog program may produce the same outputs on a laptop and on a cloud infrastructure. Clearly, a language with a logic-based semantics is in a good position for supporting this, the logic acting as an unambiguous standard of what every implementation must accomplish. However, interoperability across platforms and systems also requires technical infrastructures and language standards, which also need to be developed to realise this potential (cf. Fig. 1).

Optimisability and Performance

An implicit assumption about declarative approaches, at least on the user side, is that implementations will be able to find optimised ways of computing the requested output. Indeed, by specifying *what* should be done rather than *how*, users leave it to the system to choose the path that it considers best. Unfortunately, this is not easy, and perfect optimisation is provably impossible even for simple languages like Datalog.[5] Moreover, system implementers are faced with a dilemma: should they attempt to optimise computation, even if imperfectly, or should they leave it to users to find most efficient problem encodings? Different systems have chosen different paths here, and some allow for extra-logical optimisation hints through which users can control optimisation strategies. In practice, users do not usually ask for performance to be optimal, but to be “good enough”, which today’s mature logical reasoning systems can deliver in many relevant cases.

Explainability

With the spread of increasingly complex and functionally opaque AI systems, *explainability* has become an important objective in system design, an objective that logical approaches have been argued to realise to a large extent. Indeed, their elegant mathematical foundations often provide for high-level “explanations” for why certain outputs were produced, e.g., in the form of structured proofs. However, explainability may be required for many reasons, such as increasing user acceptance, assigning blame (accountability), or assessing a system’s true inner workings for legal compliance (Langer et al. give an insightful overview from a user perspective [42]). Logical approaches are in an excellent position to support many forms of explanation, but additional services are typically needed to truly cater for the requirements of specific scenarios.

Verifiability and Certifiability

The availability of a formal mathematical specification of the intended semantics can also help us to ensure the correctness and reliability of implementations. Two fundamental approaches in this regard are *verification* – proving that a system conforms to the mathematical specification – and *certification* – providing proofs that a particular result agrees with what the mathematical specification requires. Practical Datalog systems are highly optimised, complex systems, and we are not aware of any effort to verify such a tool in its entirety. Benzaken et al. automatically *extract* a proof-of-concept Datalog reasoner from a formalisation of Datalog in the Coq proof assistant [7]. As an alternative, Tantow et al. propose a way to

[5] For example, a Datalog program may contain a subset of rules that are redundant in the sense that the facts they contribute to the output will always be entailed by the remaining rules as well. However, checking if this is the case is undecidable. In this sense, Datalog can never be fully declarative: the details of how the user specifies the task will sometimes affect the efficiency of computation.

formally certify the correctness of the outputs of optimised Datalog engines based on a *proof checker* implemented in the proof assistant LEAN [64].

Intuitive Understandability

It has long been argued that formal logic, being developed as a model of human thought, were intrinsically easier to understand than more mechanistic specifications (see, e.g., [40]). However, conclusive empirical arguments in favour of this thesis seem to be missing. There is in fact a long-running debate regarding the agreement of formal logic with human intuition.[6] Leaving such debates aside, it seems undeniable that humans are rather good at mastering all kinds of computer languages with suitable training, so that an appeal to natural human intuition is of little practical significance for the actual utility of a language.

Conciseness and Fast Development

With most implementation details delegated to lower levels, logical specifications tend to be very short, which can translate into faster development and cheaper maintenance. Of all potential benefits listed here, this one might have the greatest force. Especially in the prototype stages of a project (which most projects never surpass), the time needed to see first results is far more important than actual performance, be it in academic research or industrial development. Logical languages are highly attractive from this perspective, since they embody unique ways of reducing complex problems to their conceptual core.

3.2 What is it Good for?

Datalog is not a general-purpose programming language, but a *domain-specific language* that is deliberately restricted to certain data manipulation tasks. The general benefits discussed in Section 3.1 can only come to bear in scenarios where the functionality of Datalog is adequate to address a relevant problem. To understand better what kind of tasks Datalog is suitable for, we review some of the ways in which Datalog has been used in research and practice.

Taking the view of a user of Datalog, we cluster use cases by their dominating computing task, not by their application domain. For example, the analysis of a medical database of patient records might be similar, on a technical level, to the analysis of customer retail data, whereas rule-based reasoning over a medical ontology may involve very different usage patterns. Our classification aims to bring out similarities in the required Datalog programs and expressive features, and in the type and provenance of the input data. Other ways of organisation would be possible, and, even under our scheme, some applications might as well be classified differently.

3.2.1 Rule-Based Knowledge Representation and Reasoning

As a logic-based language, Datalog has naturally been applied to many logical reasoning tasks. In fact, simple if-then rules come up in many applications where some form of “knowledge” is to be represented and reasoned with, including in areas that are not directly concerned with mathematical logic. For example, in computer vision, the task of 3D scene understanding may naturally involve rules such as the following: “If the pen is on the table, and the table is in the room, then the pen is in the room.” This is an example of *qualitative spatial reasoning*,

[6] A famous experiment in this area is the *Wason selection task*, which asks participants to check a given logical rule by inspecting cards; `https://en.wikipedia.org/wiki/Wason_selection_task`.

i.e., reasoning about spatial relations of objects and regions [6]. While general theories of spatial reasoning can be complex, it is possible to express many meaningful relationships in simple deterministic rules.

▶ **Example 3.** The region-connection calculus (RCC) speaks about binary relationships of regions, such as *disconnected* (DC, when two non-overlapping regions do not touch), *externally connected* (EC, when two non-overlapping regions do touch at their margins), or *non-touching proper part* (NTPP, when one region is contained in another without touching its margins). Some valid rules about these relations are the following [6]:

```
  DC(?x,?z) :- NTPP(?x,?y), EC(?y,?z) .
  DC(?x,?z) :- NTPP(?x,?y), DC(?y,?z) .
NTPP(?x,?z) :- NTPP(?x,?y), NTPP(?y,?z) .
```

It is easy to imagine how a manual implementation of the exhaustive processing of such rules can get cumbersome and error-prone. A Datalog system would be a more robust and maintainable solution.

Deterministic rule-based computations also commonly arise in research on knowledge representation and reasoning, symbolic AI, and theorem proving even when dealing with logics that are more complicated or syntactically unrelated to Datalog. Some examples:

- *Answer Set Programming* (ASP) is a powerful problem-solving paradigm that extends well beyond plain Datalog [23, 28]. Leading systems perform ASP reasoning by first *grounding* the logic program, which amounts to Datalog reasoning over a slightly rewritten program. Indeed, the modern grounders *gringo* (part of Clingo [27]) and *iDLV* (part of DLV [43]) are also among the most capable Datalog systems today.
- The W3C *Web Ontology Language* (OWL) also defines several sub-languages that have been designed with scalability in mind [52]. All of these support polynomial-time reasoning procedures that can be encoded in Datalog. For example, a sound and complete calculus for OWL EL can be expressed in a dozen Datalog rules [13], which has been exploited, e.g., to produce and compare proof structures under various proof systems [3].
- Datalog has been used to perform part of the reasoning task for certain first-order logics with arithmetic constraints, thereby speeding up reasoning overall [10]. This required a non-trivial translation of the original problem into Datalog rules.

In all of these cases, logical rules are closely related to the original format of the problem at hand, so that it is very natural to deploy Datalog reasoning in these contexts. Moreover, all of the examples are based on plain Datalog, without any extensions.[7] As we will see below, this is not typical for many other application areas.

3.2.2 Analysis of Structured Data

Datalog is well suited for recursive computations on relational structures, and as such finds applications in areas were complex structures need to be navigated and analysed. An important field where this is relevant is the analysis of software. Indeed, source code is heavily structured and this structure is significant for its functionality and quality.

[7] Admittedly, the application in theorem proving could have benefited from some arithmetic built-ins, but the necessary calculations could still be simulated through finite look-up tables [10].

▶ **Example 4.** A pioneering work in this application area has been *CodeQuest*, a Datalog-based source code query engine [31]. Relations in this case are extracted directly from Java sources. The following Datalog rules from the original paper, e.g., recursively navigate the subtype hierarchy to discover situations were one method overrides another:[8]

```
    hasSubtype(?T,?S) :- extends(?S,?T) .
    hasSubtype(?T,?S) :- implements(?S,?T) .
hasSubtypePlus(?T,?S) :- hasSubtype(?T,?S) .
hasSubtypePlus(?T,?S) :- hasSubtype(?T,?MID), hasSubtypePlus(?MID,?S) .
   overrides(?M1,?M2) :- strongLikeThis(?M1,?M2), hasChild(?C1,?M1), hasChild(?C2,?M2),
                         inheritableMethod(?M2), hasSubtypePlus(?C2,?C1) .
```

The concepts behind CodeQuest laid the foundation for the code analysis company *Semmle*, which was acquired by GitHub in 2019 to improve GitHub's own analysis of software security and vulnerabilities.[9]

Another influential project in the domain of software is *Doop*,[10] which provides facilities for pointer and taint analysis for Java programs. In contrast to CodeQuest and Semmle, these analyses investigate data and control flows in compiled Java ByteCode. Recursive Datalog rules are required, e.g., to trace data across recursive function calls. The Soufflé Datalog engine has been originally developed for this use case [36].

Applications of Datalog for advanced analytics of structured data can also be found in many other areas. For example, Piro et al. describe an analysis of real-world health records for quality assurance, for which they use a Datalog program of 174 rules on 293M distinct relational facts [58]. Another notable source of structured data are web pages, and indeed Datalog can be used to analyse the underlying tree-shaped document model (DOM). An impressive example of this approach is given by Furche et al., who report on an advanced system for the fully automated extraction of relational facts from websites [26].

3.2.3 Data Extraction and Transformation

Datalog was originally conceived as a recursive query language for relational databases [14]. However, the most common type of recursive queries are *path queries*, which test reachability along a path of relations, and such queries are meanwhile supported both by SQL (at least to a certain extent) and by dedicated graph query languages such as SPARQL. Nevertheless, Datalog is still appreciated as a language for query answering and data extraction in several applications. Examples include the Google projects Yedalog [16] and Logica [46], and the database system Datomic [17]. In these cases, Datalog is chosen not just for its expressive power, but also for its simplicity and maintainability in comparison to other query languages.

A particular advantage of Datalog in such applications is its natural capability of defining *views*, i.e., "virtual" relations (database tables) that are built from the given data. Indeed, Datalog programs declare derived relations and specify rules to populate them with suitable

[8] Most of the companion data for the publication is no longer online, but the Internet Archive holds a page that describes the predicates that the system used: `https://web.archive.org/web/20061009150649/http://progtools.comlab.ox.ac.uk/projects/codequest/projects/codequest/predicates_html`. The relations extends, implements, and hasChild in our example were parsed directly from Java, while inheritableMethod ("the method is neither private nor static") and strongLikeThis ("two elements have the same signature") were derived by rules that are no longer available.

[9] `https://github.blog/news-insights/company-news/github-welcomes-semmle/`, published 18 Sept 2019, retrieved 05 Sept 2025

[10] `https://github.com/plast-lab/doop`, retrieved 05 Sept 2025

facts. Such views can then be used as a basis for defining further views and queries, and whole data extraction and transformation pipelines can be defined in a modular, maintainable way. Following this approach, Logica employs Datalog as an abstraction layer on top of underlying SQL databases, and RDFox takes a similar route for RDF graph databases [56].

Using Datalog in data extraction often requires data transformation functions that are not part of pure Datalog, e.g., for string formatting or arithmetic calculations. Especially in database theory, researchers have also argued that *value invention* is crucial for data exchange settings [24]. This leads to *existential rules* (also known as *tuple-generating dependencies*), which we briefly discuss in Section 11. To satisfy such rules, new elements, called *named nulls*, might be introduced during reasoning. In practical settings, many potential applications of this feature can also be realised by using specific values in a concrete datatype instead.

3.2.4 Graph and Network Analysis

Graphs are a model application field for iterative algorithms and recursive processing, so Datalog seems to be a natural candidate for tasks in this space. Indeed, many types of graph structures have found important and prominent applications, from basic *networks* (where most information is in the global connectivity structure) to rich *graph databases* (that contain labelled edges, data values, and many further structures). The latter are typically used like other complex databases, and related applications are as discussed in Section 3.2.3. For simple graphs and network structures, however, there are many specific algorithms and analysis methods that one might try to realise with Datalog.

In the area of network analysis, many algorithms are iterative computations, e.g., for producing centrality measures that can be used in recommender systems. A well-known algorithm of that type is *PageRank*, which is computed by propagating weights along edges of a graph, not entirely unlike a weighted version of transitive closure. Datalog systems that have been designed to support such computations include SocialLite [62], EmptyHeaded [1], and Dynalog [21]. This requires some form of external termination criterion, usually by specifying how often rules can be applied, since such algorithms converge to the exact solution without reaching it exactly in finite time.

Many classical graph algorithms are exact algorithms that do not require such approximation. Examples include Dijkstra's algorithms for the computation of shortest paths or Tarjan's algorithm for finding strongly connected components. We should not expect that Datalog can re-produce all details of such algorithms, e.g., the specific data structures used to organise the processing, but we might still hope for an elegant declarative formulation. Unfortunately, this is not easy to obtain, since such algorithms often include aspects of (don't care) non-determinism, (monotonic) recursive optimisation, and dynamic programming – all of which turn out to be somewhat problematic for standard Datalog.

▶ **Example 5.** Consider a graph with edges that have numeric weights > 0. Shortest weighted paths from a given source vertex start can be found by performing a depth-first search that maintains the least cost required to reach a node until all costs are stable. In Datalog, one could express this as follows, using a `#min` aggregate (see Section 8):

```
                   reach(?t,?cost) :- edge(start,?t,?cost) .
reach(?t,#min(?bestCost+?cost)) :- reach(?v,?bestCost), edge(?v,?t,?cost) .
```

Conceptually, this would be correct, but most current Datalog systems do not support such recursive uses of aggregation, since this can lead to non-terminating or undefined behaviour. We will return to this case in Example 43 later on. As we will see there, the problem can also be solved without recursive aggregation, but only by computing *all* costs under which each vertex can be reached, which can involve infinitely many inferences (and exponentially many even in acyclic graphs). This approach is therefore not practical.

Systems that target use cases in graph and network analysis have therefore included support for recursive aggregation as well, at least for extrema aggregates (min, max). In most cases, the responsibility for using this in meaningful ways lies with the programmer, although some works have proposed conditions under which a well-defined declarative solution is guaranteed to exist (see Section 8). For now, we can therefore conclude that graph and network algorithms are a promising and important field of application for Datalog, but that special extensions are required to support it.

4 Pure Datalog

After the applied perspective taken above, we are going to approach Datalog from the viewpoint that is typical in research, introducing the relevant terminology and underlying mathematical concepts. Equipped with the necessary language, we will find it easy to specify the semantics of Datalog, and we will do so in four different but equivalent ways.

4.1 Datalog Terminology and Abstract Syntax

Our syntax follows the notational conventions of predicate logic, which is usually preferred in research works and textbooks on Datalog. Alternative notations and common variations are discussed below. In contrast to the formats in Fig. 1, Datalog research uses an *abstract* syntax that does not bother with practical concerns of unambiguous parsing. We therefore assume that relevant syntactic elements are given in disjoint sets (alphabets), instead of constructing (and recognising) them from individual characters.

▶ **Definition 6** (Terms, Atoms, Rules, and Facts)**.** *Datalog is based on countably infinite, mutually disjoint alphabets* $\mathbf{P}$ *of* predicate symbols*,* $\mathbf{C}$ *of* constant symbols*, and* $\mathbf{V}$ *of* variables*. A* term t *is either a constant or a variable, i.e.,* $t \in \mathbf{C} \cup \mathbf{V}$*. Every predicate symbol* $p \in \mathbf{P}$ *is associated with a unique* arity $\mathsf{ar}(p) \in \mathbb{N}$ *(we allow arity* 0*).*

An atom *is an expression of the form* $p(t_1, \ldots, t_\ell)$ *for a predicate* $p \in \mathbf{P}$ *and terms* $t_1, \ldots, t_\ell$ *with* $\ell = \mathsf{ar}(p)$*. List of terms are denoted in bold, e.g.,* $\boldsymbol{t} = t_1, \ldots, t_{|\boldsymbol{t}|}$*, so we may write the previous atom as* $p(\boldsymbol{t})$*. Similarly,* $\boldsymbol{x}$*,* $\boldsymbol{y}$*, etc. are used to denote lists of variables. A logical expression is* ground *if it does not contain variables.*

A Datalog rule *is an expression of the form* $H \leftarrow B_1, \ldots, B_n$*, where* $B_1, \ldots, B_n$ *(the* body*) and* H *(the* head*) are atoms. If* $n = 0$*, we omit* $\leftarrow$ *and write the rule as a* generalised fact H*. A* ground fact *(or simply* fact*) is a generalised fact that is ground.*

Our syntax therefore agrees with our initial example (1). Some authors stay even closer to the logical legacy of Datalog, and make conjunction and universal quantification explicit, writing rules as $\forall \boldsymbol{x}.B_1 \wedge \cdots \wedge B_n \rightarrow H$, where $\boldsymbol{x}$ is a list of all variables that occur in the rule. Here, we also used the forward implication $\rightarrow$ that is more common in mathematical logic. We trust that readers can easily adjust to any of these conventions.

▶ Note 7 (Atoms vs. Facts). According to Definition 6, facts and ground atoms are syntactically identical. The difference is conceptual. We speak of *atoms* when referring to syntactic (sub)structures in logical expressions, and of *facts* when referring to logical assertions that are stated to hold true.

▶ Note 8 (Safety). A common requirement in many works is that Datalog rules are *safe* in the sense that every variable that occurs in the head must also occur in the body. For a counterexample, consider the rule $\mathsf{greaterOrEqual}(x, y) \leftarrow \mathsf{largest}(x)$, which expresses that the largest element is bigger than anything else. In other words: if we find an element e for which $\mathsf{largest}(e)$ holds, the rule should entail $\mathsf{greaterOrEqual}(e, f)$ for *every* element f. The semantics of the rule therefore depends on our choice of what elements f exist, which is not defined in Datalog. We can always consider arbitrary elements $f \in \mathbf{C}$, but then the rule would produce infinitely many inferences. Alternatively, we might restrict to the (finitely many) elements that are actually mentioned in the given input data or Datalog rules under consideration. While finite, this has the disadvantage that the result produced by the rule may change if an element gets mentioned in a seemingly unrelated context. Safety prevents these problems by ensuring that all head variables are constrained by concrete atoms in the body. In database theory, queries that are not affected by assumptions on the overall set of domain elements are called *domain independent* [2].

Nevertheless, some modern systems do not require safety. For example, Logica [46] allows unsafe rules and even facts, such as $\mathsf{successor}(x, x + 1)$ (see Section 7 for more details on arithmetic functions in Datalog). The system ensures, however, that *output predicates* are safe (domain independent), whereas potentially infinite predicates such as successor can only be used as "virtual" intermediate results that are not returned.

▶ Note 9 (Named vs. Unnamed Perspective). In relational databases, the arguments (columns) in relations are generally *named* by suitable attribute names. Predicates then are characterised by a list of attribute names rather than by an arity, and atoms mention attribute names to refer to parameters. Syntactically, this might look like this: $\mathsf{date}(\mathsf{year} : 2025, \mathsf{month} : 9, \mathsf{day} : 25)$. In contrast, Definition 6 uses an *unnamed* perspective where the order of parameters in an atom determines their role in a relation (as in $\mathsf{date}(2025, 9, 25)$). In practice, the named perspective offers increased readability, in particular since we do not need to mention unused parameters. The following rules illustrate this point (assuming a built-in $\leq$; see Section 7):

$$\mathsf{millenial(x)} \leftarrow \mathsf{birthdate}(x, y, z, v), 1981 \leq y, y \leq 1996 \tag{2}$$
$$\mathsf{millenial(person\!: x)} \leftarrow \mathsf{birthdate}(\mathsf{person} : x, \mathsf{year} : y), 1981 \leq y, y \leq 1996 \tag{3}$$

Logic research and traditional logic programming prefers the unnamed perspective, but modern Datalog systems may support either variant (and sometimes both). Since attribute names are a fixed part of the schema (not of the data), it is always possible to unambiguously translate from one perspective to the other.

▶ Note 10 (Conjunctions in Heads, Disjunctions in Bodies). Datalog is deterministic by design, and disjunctive information is therefore not commonly supported. However, disjunctions in rule bodies are syntactic sugar, since a rule $H \leftarrow B_1 \vee B_2$ could equivalently be expressed in two rules $H \leftarrow B_1$ and $H \leftarrow B_2$ (even if B_1 and B_2 are conjunctions of many atoms). Likewise, conjunctions in heads can be simulated by writing $H_1 \wedge H_2 \leftarrow B$ as $H_1 \leftarrow B$ and $H_2 \leftarrow B$. Both syntactic features are supported by many systems. Typically, head conjunction is expressed using comma as in rule bodies (e.g., in Soufflé, RDFox, and Nemo), while body disjunction may use ; as in Prolog (e.g., in Soufflé) or | (e.g., in Logica).

▶ **Definition 11** (Datalog Programs). *A* Datalog program *is a triple* $\langle P, \mathbf{P}_{\mathsf{in}}, \mathbf{P}_{\mathsf{out}} \rangle$*, where*
- P *is a finite set of rules,*
- $\mathbf{P}_{\mathsf{in}}$ *is a non-empty set of* input predicates *that do not occur in rule heads of* P*, and*
- $\mathbf{P}_{\mathsf{out}}$ *is a non-empty set of* output predicates.

We require all predicates in $\mathbf{P}_{\mathsf{in}}$ *and* $\mathbf{P}_{\mathsf{out}}$ *to occur in* P*. Input predicates are also called* EDB predicates*, and all other (non-input) predicates are also called* IDB predicates.[11]

We generally assume that input and output predicates are part of every program. Intuitively, input predicates denote relations that are provided as an input for the Datalog program whereas output predicates denote those that are considered the result of the computation. If the specific sets are irrelevant or clear from the context, we may simply write Datalog programs as P.

▶ Note 12 (EDB and IDB vs. Input and Output). Input (=EDB) predicates can always be taken to be the predicates that occur only in rule bodies, and IDB predicates are the rest. Regarding output predicates, there is less agreement in the literature on Datalog. Some works have not made outputs explicit at all, and instead consider additional *queries* on top of Datalog programs to access the results of a computation [2]. Some other authors only allow for a single output relation that is an explicit part of Datalog queries, which thereby correspond more closely to the result of a standard database query Our formalisation with multiple output predicates largely agrees with Dantsin et al. who likewise define an input and output schema [19]. As discussed in Section 3.2, there are meaningful applications where more than a single output relation is wanted.

In any case, specifying the intended outputs (as part of the program or as an additional query) is important to clarify the semantics of Datalog, since it distinguishes facts that are considered part of the actual result from intermediate results. Programs with very different non-output predicates may still produce the same output, and systems likewise can optimise computations as long as the same outputs are produced.

The input that a Datalog program is evaluated on is an arbitrary set of facts for the input predicates. Following standard terminology, we call such sets *databases.*

▶ **Definition 13** (Databases). *A* database $\mathcal{D}$ *is a set of facts.* $\mathcal{D}$ *is an* input database *for a Datalog program* $\langle P, \mathbf{P}_{\mathsf{in}}, \mathbf{P}_{\mathsf{out}}\rangle$ *if it only uses predicates from* $\mathbf{P}_{\mathsf{in}}$*. Similarly, an* output database *is one that only uses predicates from* $\mathbf{P}_{\mathsf{out}}$*.*

Using this terminology, we find that a Datalog program specifies a function that maps input databases to output databases. The exact definition of this function is the topic of the next Section 4.2.

▶ Note 14 (Infinite Databases). It is common to further restrict databases to be finite. Definition 13 does not require this, since we will be considering some cases where infinite outputs may (mathematically) follow. In practice, databases are of course finite.

4.2 Datalog Semantics Four Ways

Next, we specify the semantics of Datalog in more detail by defining what exactly the output database should be on a given program and input. In fact, rather than just one definition, we give four. The mutual agreement of these different perspectives should be taken as further evidence that Datalog is defined in *the right way*, however we choose to look at it.

4.2.1 Operational Semantics: The Direct Consequence Operator

Possibly the most intuitive way of defining the outputs of a Datalog program is to think of its rules as being “applied” repeatedly to the given facts until no new inferences are obtained.

[11] This traditional terminology comes from *extensional database* and *intensional database.*

To make this formal, we first define the important notion of a substitution.

▶ **Definition 15** (Substitution). *A substitution σ is a partial mapping from variables to terms. A substitution is* ground *if it maps to constants only. For a term t, we write $t\sigma$ to denote $\sigma(t)$ if t is a variable for which σ is defined, and to denote t otherwise (in particular, the second case occurs if t is a constant). For arbitrary syntactic expressions E, the expression $E\sigma$ is obtained by replacing each term t in E with $t\sigma$.*

▶ **Definition 16** (Rule Application). *Consider a database $\mathcal{D}$ and a Datalog rule ρ of the form $H \leftarrow B_1, \ldots, B_n$. A ground substitution σ is a* match *for ρ on $\mathcal{D}$ if (1) σ is defined exactly on the variables in ρ, and (2) $B_1\sigma, \ldots, B_n\sigma \in \mathcal{D}$. In this case,* applying *ρ to $\mathcal{D}$ under σ yields the inference $H\sigma$.*

If rules are safe (see Note 8), then matches necessarily map to constants in $\mathcal{D}$; otherwise, one must fix a domain for the constants that may be used. The most common way to define the operational semantics of a Datalog program P is through the *immediate consequence operator T_P*, which performs all possible rule applications in parallel.[12] Iterating this process leads to the output.

▶ **Definition 17** (Operational Datalog Semantics). *Consider a Datalog program $\langle P, \mathbf{P}_{\mathsf{in}}, \mathbf{P}_{\mathsf{out}} \rangle$. For a database $\mathcal{D}$, let $T_P(\mathcal{D}) = \{H\sigma \mid \text{there is } H \leftarrow B_1, \ldots, B_n \in P \text{ with match } \sigma \text{ on } \mathcal{D}\}$.*

Given an input database $\mathcal{D}$ for P as above, we define $T_P^0 = \mathcal{D}$ and $T_P^{i+1} = T_P(T_P^i) \cup \mathcal{D}$ for all $i \geq 0$. Moreover, let $T_P^\infty = \bigcup_{i\geq 0} T_P^i$. The output database of P over $\mathcal{D}$, denoted $P(\mathcal{D})$, is the restriction of T_P^∞ to output predicates, i.e., the set $\{p(c_1, \ldots, c_n) \mid p(c_1, \ldots, c_n) \in T_P^\infty, p \in \mathbf{P}_{\mathsf{out}}\}$.

The definition of T_P^∞ seems to suggest that it could be an infinite union. However, if rules are safe or if a finite domain is used with unsafe rules, then it is actually always a finite set, and hence can be obtained as a finite union. Moreover, it is not hard to see that $T_P^i \subseteq T_P^{i+1}$ for all $i \geq 0$, since previous inferences will be reliably re-derived in every future step. Therefore, $T_P^\infty = T_P^k$ for some number $k \geq 0$, and a reasoner could simply iterate the computation of T_P^i until a *fixed point* is reached in the sense that $T_P^{i+1} = T_P(T_P^i) \cup \mathcal{D} = T_P^i$. It turns out that T_P^∞ is actually the *least fixed point* of T_P that contains $\mathcal{D}$: the $\subseteq$-smallest set $\mathcal{I}$ such that $\mathcal{D} \subseteq \mathcal{I}$ and $T_P(\mathcal{I}) = \mathcal{I}$.

▶ **Example 18.** We return to the family-related example program from Listing 1. In practice, the distinction between input databases and rules can be blurred, since facts can be equivalently treated as special rules. Here, we will assume that the input database $\mathcal{D}$ consists of the five facts in L10–L12.

Initially, we have $T_P^0 = \mathcal{D}$. To compute T_P^1, we consider all rule applications that are possible on T_P^0. For example, rule L15 can be applied for the match $\{\texttt{?child} \mapsto \mathsf{alice}, \texttt{?father} \mapsto \mathsf{bob}\}$. Each of the initial facts gives rise to one match, and we obtain $T_P^1 = T_P^0 \cup \{\mathsf{parent(alice, bob)}, \mathsf{parent(alice, cho)}, \mathsf{parent(cho, daniel)}, \mathsf{parent(cho, eiko)}, \mathsf{parent(finley, eiko)}\}$. Next, applying rule L17, we can derive $T_P^2 = T_P^1 \cup \{\mathsf{ancestor(alice, bob)}, \mathsf{ancestor(alice, cho)}, \mathsf{ancestor(cho, daniel)}, \mathsf{ancestor(cho, eiko)}, \mathsf{ancestor(finley, eiko)}\}$. Now, rule L18 is applicable, and we obtain $T_P^3 = T_P^2 \cup \{\mathsf{ancestor(alice, daniel)}, \mathsf{ancestor(alice, eiko)}\}$, and finally, by rule L21, we get $T_P^4 = T_P^3 \cup \{\mathsf{commonAnc(eiko)}\}$. No further derivations are possible, hence $T_P^5 = T_P^4 = T_P^\infty$.

[12] The notation T_P dates back to logic programming in Prolog; T stands for *truth*.

To avoid a dependency on the choice of domain, we state the next result for safe Datalog.

▶ **Theorem 19.** *Let P be a safe Datalog program with an input database $\mathcal{D}$. The size of T_P^∞ (and therefore also of $P(\mathcal{D})$) is polynomially bounded in the size of $\mathcal{D}$, and exponentially bounded in the size of P and $\mathcal{D}$.*

Deciding if a fact $p(c_1, \ldots, c_n)$ is in T_P^∞ is PTIME*-complete in the size of $\mathcal{D}$ (data complexity), and* EXPTIME*-complete overall.*

Proof sketch. For the upper bounds on the size of T_P^∞, note that, if we use d different constants, then there are at most $d^{\mathsf{ar}(p)}$ distinct facts for a predicate p. The number d is linear in $\mathcal{D}$ and P. The number and arity of predicates that need to be considered is linear in P, but the arity figures as an exponent in the upper bound. This yields the claimed exponential bound. If P is fixed and only $\mathcal{D}$ can grow, then predicates (and arities) are fixed, and only the number of constants d can grow, leading to the claimed polynomial bound.

For the complexity upper bounds, one can make a similar argument about the number of possible rule matches (exponentially many overall, polynomially many w.r.t. $\mathcal{D}$). Since we can naively compute T_P by considering each possible match, the direct computation of T_P^i for larger and larger i is possible in deterministic exponential (resp. polynomial) time. The matching lower bounds (hardness) require some more work, e.g., by simulating a time-bounded Turing machine computation in Datalog (see [19] for details). ◀

4.2.2 Model-Theoretic Semantics: The Least Model

The primary way of defining consequences in mathematical logic is model theory: one first defines what the *models* of a logical theory T are, and then declares logical consequences of T to be exactly those statements that hold true in all models of T. This general idea also works for Datalog, and the required notion of model is very simple.

▶ **Definition 20** (Model-Theoretic Datalog Semantics). *Consider a Datalog program P with input database $\mathcal{D}$. An* Herbrand model[13] *of P and $\mathcal{D}$ is a database $\mathcal{H}$ such that*

1. *$\mathcal{D} \subseteq \mathcal{H}$, and*
2. *for every rule $\rho \in P$ of the form $H \leftarrow B_1, \ldots, B_n$, and every match σ for ρ over $\mathcal{H}$, we also have $H\sigma \in \mathcal{H}$.*

The output database of P over $\mathcal{D}$ is the set of all output facts that are true in all Herbrand models, i.e., the set $\{p(c_1, \ldots, c_n) \mid p \in \mathbf{P}_{\mathsf{out}}$ and $p(c_1, \ldots, c_n) \in \mathcal{H}$ for all Herbrand models $\mathcal{H}$ of P and $\mathcal{D}\}$.

▶ **Example 21.** Consider again the family example from Listing 1. The set T_P^∞ from Example 18 is a Herbrand model: our exhaustive application of T_P ensures that all rules are satisfied as required by Definition 20 (2). However, there are also other Herbrand models, in particular the set of *all* possible facts that can be expressed with the predicates mother, father, parent, ancestor, and commonAnc and the constants alice, bob, cho,eiko, and finley.

The difference of Herbrand models and standard models of first-order predicate logic is that Herbrand models fix the domain of interpretation to be exactly a set of constants, such that (syntactic) constants agree with (semantic) domain elements. This is why we can view Herbrand models are sets of (syntactic) facts.

[13] Named after French mathematician *Jacques Herbrand*, natively pronounced with the *H* silent. As language evolves, the concept might detach from the person – typically accompanied by lower-case lettering as in *boolean* or *cartesian* – which might justify an anglicised pronunciation: *a* herbrand model.

The semantics of Definition 17 and 20 agree, and it turns out that we only need to consider one specific *least* Herbrand model:

▶ **Theorem 22.** *The set of all Herbrand models of a Datalog program and input database has a least element, and this least Herbrand model coincides with* T_P^∞.

Proof sketch. The claim is shown in two parts: (1) T_P^∞ is an Herbrand model, and (2) every fact in T_P^∞ is necessarily in every Herbrand model. Part (1) is clear by contradiction: if it were no model, it would violate Definition 20 (2), but then it would not be a fixed point of T_P either. Part (2) is easy to show separately for each T_P^i by induction over $i \geq 0$. ◀

This is a very pleasing result, since it connects an elegant logical semantics with the more implementable operational view. In fact, the agreement would even hold if we would consider arbitrary first-order models instead of Herbrand models for Definition 20, since only facts about the given constants can be true across all such models. Using Herbrand models is closer to the understanding of logic programming and databases, however, where syntactic constants usually have a direct (known) relationship to domain elements.[14]

4.2.3 Proof-Theoretic Semantics: Proof Trees

Besides model theory, logic has another way of defining semantics: a logical consequence is what can be proven. In the case of Datalog, there is a natural notion of "proof" that can be described directly.

▶ **Definition 23** (Proof-Theoretic Datalog Semantics)**.** *Consider a Datalog program P with input database $\mathcal{D}$. A* proof tree *with respect to P and $\mathcal{D}$ is a tree structure T that satisfies the following conditions:*
1. *every node n of T is labelled by a fact $\lambda(n)$,*
2. *if n is a leaf node, then $\lambda(n) \in \mathcal{D}$,*
3. *if n is an inner node, then n is additionally labelled by a rule $H \leftarrow B_1, \ldots, B_k \in P$ and a substitution σ, such that (1) $\lambda(n) = H\sigma$ and (2) n has exactly k child nodes $c_1, \ldots, c_k$ with $\lambda(c_i) = B_i\sigma$ for all $1 \leq i \leq k$.*

If a proof tree T has root node r, we say that T is a proof for $\lambda(r)$ with respect to P and $\mathcal{D}$. The output database of P over $\mathcal{D}$ is the set of all output facts for which there is a proof tree, i.e., the set $\{p(c_1, \ldots, c_n) \mid$ there is a proof tree for $p(c_1, \ldots, c_n)$ with respect to P and $\mathcal{D}\}$.

▶ **Example 24.** Proofs trees can be inutitively visualised. For example, Fig. 3 depicts a proof for commonAnc(eiko) for the program from Listing 1. Here, the rules that label inner nodes in Definition 23 are shown as separate nodes (with blue boundaries), whereas substitutions σ are implicit. In Nemo Web, similar trees can be obtained by pressing the magnifier glass icon next to any output fact. Note that the same inference may admit several proofs. By default, Nemo returns that proof that corresponds best to the steps that were used to produce the inference. In logic programming, the specific proof that was followed by a system is also known as the *trace*. When used in the command line, Nemo can also produce traces in machine-readable formats (e.g., JSON) for further processing. Few Datalog engines provide this feature, with Soufflé being another notable example (command-line, no visualisation).

We obtain the expected correspondence with the previous definitions of Datalog semantics.

[14] Logic programming sometimes works with a *Unique Name Assumption* where each domain element has exactly one syntactic denotation.

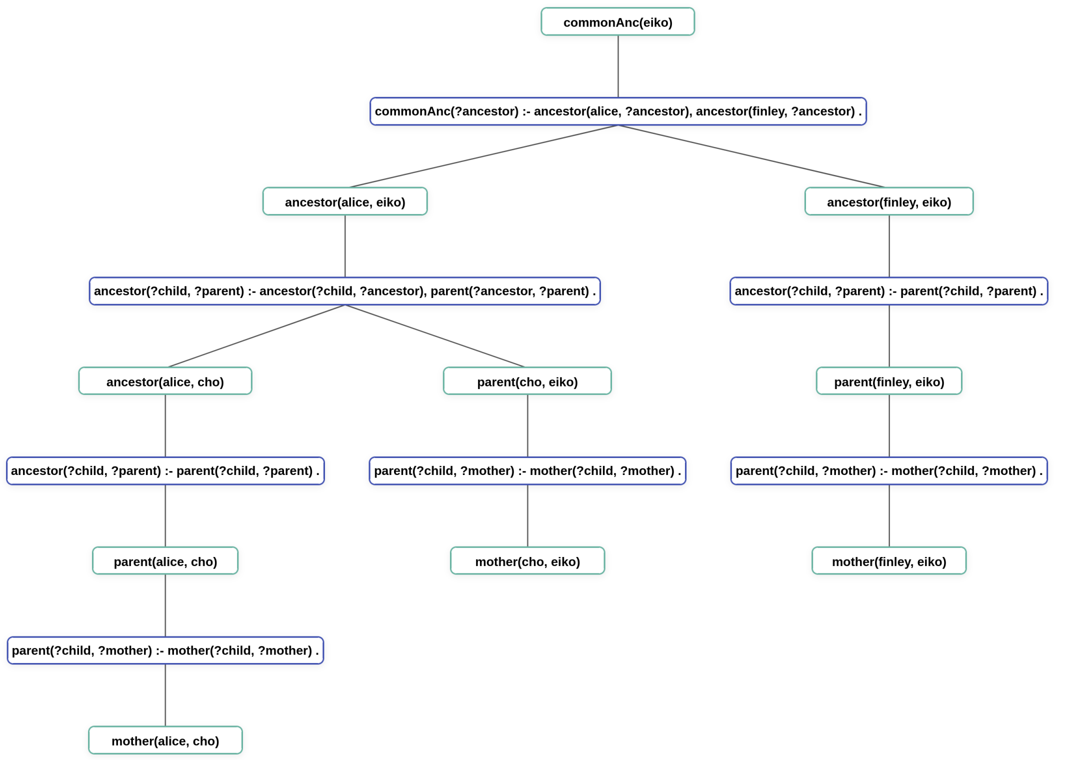

commonAnc(eiko)
commonAnc(?ancestor) :- ancestor(alice, ?ancestor), ancestor(finley, ?ancestor) .
ancestor(alice, eiko)
ancestor(finley, eiko)
ancestor(?child, ?parent) :- ancestor(?child, ?ancestor), parent(?ancestor, ?parent) .
ancestor(?child, ?parent) :- parent(?child, ?parent) .
ancestor(alice, cho)
parent(cho, eiko)
parent(finley, eiko)
ancestor(?child, ?parent) :- parent(?child, ?parent) .
parent(?child, ?mother) :- mother(?child, ?mother) .
parent(?child, ?mother) :- mother(?child, ?mother) .
parent(alice, cho)
mother(cho, eiko)
mother(finley, eiko)
parent(?child, ?mother) :- mother(?child, ?mother) .
mother(alice, cho)

Figure 3 Proof tree visualisation in Nemo Web, based on the program from Listing 1

▶ **Theorem 25.** *T_P^∞ is exactly the set of facts for which there is a proof tree.*

Proof sketch. In one direction, we find that every node label in every proof tree is in T_P^∞. This can be shown inductively, starting from trees of smaller height, where one-node trees are leafs labeled by facts from $\mathcal{D}$. The result for larger trees follows by using the induction hypothesis to show that the substitution σ from Definition 23 (3) is a match for the given rule.

In the other direction, it is similarly straightforward to construct proofs from the sequence of rule applications that was used to infer a fact in T_P^∞. Indeed, $T_P^{i+1} \setminus T_P^i$ contains exactly those facts that have a proof tree of height $i + 1$ (if we assign height 0 to single-node proofs) but no proof tree of smaller height. ◀

Proof trees are adequate answers to the question why and how a particular fact has been inferred, and are thereby an important approach towards explanation and certification (see Section 3.1).

4.2.4 Logical Semantics: Datalog as Second-Order Logic

The above model theory may give the impression that Datalog is essentially a fragment of first-order logic. Indeed, as remarked in Section 4.2.2, if we consider Datalog rules as standard logical implications, then the first-order *entailments* of a Datalog program and input database coincide with the inferences defined in our other semantics. However, this view fails to capture how Datalog is commonly used: as a language to *define* output relations, with non-output relations playing only the role of intermediate auxiliary results that are useful on the path towards the desired result.

▶ **Example 26.** To illustrate, consider the left and the right Datalog program below, where c is a constant:

$$\begin{aligned} \mathsf{reach}(x,y) &\leftarrow \mathsf{edge}(x,y) \\ \mathsf{reach}(x,z) &\leftarrow \mathsf{reach}(x,y), \mathsf{reach}(y,z) \\ \mathsf{output}(z) &\leftarrow \mathsf{reach}(c,z) \end{aligned} \qquad \begin{aligned} \mathsf{output}(y) &\leftarrow \mathsf{edge}(c,y) \\ \mathsf{output}(z) &\leftarrow \mathsf{output}(y), \mathsf{edge}(y,z) \end{aligned}$$

If edge is the only input predicate and output is the only output, then these two implementations will produce exactly the same results on all input databases. Nevertheless, if we consider the programs as first-order logical theories (obtained as a conjunction of all rules in each program, with universal quantifiers added for every variable), then these theories are clearly not logically equivalent. The first-order semantics in this sense does not agree with our intended semantics.

Similar mismatches with first-order logic also appear when introducing negation (Section 6), since first-order entailment reasoning does not allow us to conclude that any fact is certainly not true, whereas Datalog is usually considered to define output relations that consist *exactly* of the entailed facts with all other facts being false. To capture this intention in logic, we can use second-order logic, which allows us to quantify over relations. In this perspective, all IDB predicates (output or not) become second-order variables. A fully formal introduction of second-order logic is beyond the scope of this lecture, but we can illustrate the idea with an example.

▶ **Example 27.** The left program in Example 26 corresponds to the following second-order logic formula. We use upper-case names to distinguish second-order variables from first-order relations.

$$\begin{matrix}\forall\,\mathsf{Reach},\\ \mathsf{Output}.\end{matrix} \left(\left(\begin{array}{lrl} (\ \ \forall x,y. & \mathsf{edge}(x,y)) & \to \mathsf{Reach}(x,y)) \land \\ (\forall x,y,z.\ \mathsf{Reach}(x,y) \land & \mathsf{Reach}(y,z) & \to \mathsf{Reach}(x,z)) \land \\ (\quad\ \ \forall z. & \mathsf{Reach}(c,z) & \to \mathsf{Output}(z)) \end{array} \right) \to \mathsf{Output}(v) \right)$$

Let us call this formula F. Then F is a formula of second-order logic with one free variable v. The innermost part of F is the conjunction of all former Datalog rules, now written as universal logical implications (we use a vertical layout for clarity). This conjunction is the premise in an outer implication that has Output(v) as its consequence. Intuitively, F can be read as saying: "The value assigned to v is such that, for every interpretation of the IDB predicates Reach and Output that satisfies all of the Datalog rules, Output(v) must be true." Universally quantifying over all relations for Reach and Output then corresponds to the quantification over all models in Section 4.2.2.

We are now interested in constants d for which $\mathcal{D} \models F[v \mapsto d]$ holds, i.e., under which the formula F with v replaced by d is true in *the interpretation* $\mathcal{D}$. The input database therefore now plays the role of a logical interpretation (model) under which a formula may or may not be true. The set of all d for which this is the case is a binary relation: exactly the relation output produced as a result of the original Datalog program.

This construction allows us to logically define relations, and the corresponding second-order formula could be used in more complex logical expressions, e.g., to check if a pair is *not* in an output relation. This is not possible in pure first-order logic. For example, it is a well-known fact that first-order logic cannot define the transitive closure of a binary relation, which is easy in Datalog (as Reach in Example 26 shows).

5 Hands-On: Importing Online Data

In Section 2, input data was given in the form of facts in the Datalog program itself. This is practical for small, static data, but not for datasets with millions of facts or managed in external systems. Many Datalog systems therefore support data imports from other sources, typically defined by adding suitable importing directives to Datalog programs. In Nemo, data can come from files (e.g., in CSV format) or from online data services. Files in turn may be local or online (to use local files in the Web application, they first need to be added through the user interface). Next, we will scale up our earlier family example to use real data from Wikidata [69].

The free knowledge base Wikidata is a sister project of Wikipedia, and collects general knowledge the whole range of encyclopaedic topics. To use this information in Nemo, we take advantage of the Wikidata Query Service [49], which lets us retrieve real-time answers for queries in the W3C SPARQL query language [32]. Familiarity with SPARQL or Wikidata is not required here, but some general aspects should be clarified.

First, Wikidata is a multilingual knowledge base that uses language-independent identifiers (so-called QIDs) for all concepts. For example, the QID of the city of Istanbul is Q406, whereas the Reasoning Web summer school is Q105700533. To find QIDs, one can simply use the search function on wikidata.org, or follow the links found on any Wikipedia article ("Wikidata item" under "Tools"). When querying Wikidata, we will mainly use IDs, formatted as IRIs such as `http://www.wikidata.org/entity/Q406`. Human-readable labels are fetched separately, mainly for display.

Second, Wikidata stores data by assigning property values to items, where properties also are identified by dedicated ids. For example, on item Q406, the property *head of government* (P6) has the value *Ekrem İmamoğlu* (Q59313241). For our purposes, we will need properties P22 (*father*) and P25 (*mother*). Viewing item pages on Wikidata is the easiest way to find out what data is available and how it is encoded.

The re-designed version of our family example can be seen in Listing 2 and online under `https://tud.link/wkrajt`. In this version, the program finds all known common ancestors of *Ada Lovelace* and *Moby* (a non-empty set!). The actual Datalog rules in L56–L61 are similar to those in Listing 1. The main difference is that we now start with two explicit elements (L56–L57), and that we need to fetch the name of each common ancestor in the end (L60). The elements we start with are defined as program-wide *parameters* (L37–L38), set to the ids of our persons of interest, which we abbreviated using a *prefix* (L35). Parameters are useful here since we need the terms in several rules, and since it is convenient to define them at the top for easy modification. Parameters can also be set when calling Nemo from the command line, so that one Nemo program can be used for various retrieval tasks.

The import of information from Wikidata is defined in L41–L53. We declare two imported predicates, `wdParent` and `wdLabel`. Both have arity two, which is derived from the number of variables selected in the SPARQL queries. The imports each declare an *endpoint* (the URL of the query service) and a *query* (as multi-line string enclosed in `"""`). The second query uses a Wikidata-specific way of fetching labels based on a virtual *service* [49].

Semantically, these statements mean that the two predicates represent the complete result of the two queries. However, Nemo does not actually run the queries as stated, but evaluates such imported predicates on demand. Therefore, instead of downloading all (>1.8M) parent-relations, only those needed for applications of rules L56–L58 are fetched. To do this, Nemo will augment the query with bindings provided from partial matches of the respective rule body. This is also important for rule L60, since it is not practically feasible to download

```
% Prefixes help to abbreviate long identifiers or URLs:
@prefix wdqs: <https://query.wikidata.org/> .
@prefix wd: <http://www.wikidata.org/entity/> .
% Parameters can be used for fixed terms throughout the program:
@parameter $personId1 = wd:Q7259 . % Ada Lovelace
@parameter $personId2 = wd:Q14045 . % Moby

% Import predicate "wdParent" (mother or father) through SPARQL:
@import wdParent :- sparql{
  endpoint=wdqs:sparql,
  query="""PREFIX wdt: <http://www.wikidata.org/prop/direct/>
    SELECT ?child ?parent WHERE { ?child (wdt:P22|wdt:P25) ?parent }"""
} .
% Import predicate "wdLabel" (English label) through SPARQL:
@import wdLabel :- sparql{
  endpoint=wdqs:sparql,
  query="""PREFIX wikibase: <http://wikiba.se/ontology#>
    SELECT ?qid ?qidLabel WHERE {
      SERVICE wikibase:label {
        <http://www.bigdata.com/rdf#serviceParam> wikibase:language "mul,en" } }"""
} .

% Find relevant ancestors, starting from selected persons:
ancestor($personId1, ?parent) :- wdParent($personId1, ?parent) .
ancestor($personId2, ?parent) :- wdParent($personId2, ?parent) .
 ancestor(?person, ?ancestor) :- ancestor(?person, ?x), wdParent(?x, ?ancestor) .
% Find common ancestors, and determine their names:
commonAnc(?qid, ?name) :- ancestor($personId1, ?qid), ancestor($personId2, ?qid),
                          wdLabel(?qid,?name) .
% Select one output predicate:
@output commonAnc .
```

Listing 2 Nemo Datalog program to find the common ancestors of Ada Lovelace and Moby

all labels through the query service (at the time of this writing, there are 634,823,210 labels in Wikidata). Nevertheless, Nemo ensures that the computed outputs are not affected by the internal choice of query strategy. In particular, if one were to return imported predicates as output, then all data would have to be fetched.

A general consideration when using SPARQL and Datalog in combination is that SPARQL itself is fairly powerful, so that one may have a choice of performing some operations in import queries instead of using Datalog rules. In fact, the program in Listing 2 can equivalently be expressed in a single SPARQL query of just a few lines. However, this query reliably runs into a timeout on Wikidata's query service, whereas the Datalog-driven query approach completes in under a minute. Another reason for using Datalog instead of SPARQL is that this allows us to combine data from different sources, as shown in a later hands-on in Section 9.

As an exercise, readers can try to extend the program by fetching additional information, such as the date of birth (P569) for every ancestor before it is returned. As a bigger variation, one can create a program that fetches all of a person's blood relatives (ancestors and all of their descendants). Try it on Johann Sebastian Bach (Q1339).

6 Stratified Datalog: Computation in Layers

Pure Datalog is too limited for many purposes, and many further features have therefore been considered for extending it. This comes at a risk: the elegant semantic characterisations discussed in Section 4.2 may fail, and in the worst case declarativity and all the associated benefits (Section 3.1) may be lost.

▶ **Example 28.** To illustrate the possible problems, let us assume that we would like to extend Datalog with a negation operator ¬ that can be used in front of body atoms to check if the atom does *not* hold true. The following program then asserts that an underage person is any human who is not an adult, and vice versa:

$$\mathsf{underage}(x) \leftarrow \mathsf{human}(x) \wedge \neg\mathsf{adult}(x) \tag{4}$$
$$\mathsf{adult}(x) \leftarrow \mathsf{human}(x) \wedge \neg\mathsf{underage}(x) \tag{5}$$

With human(alice) as the only input fact, what should the semantics of this program be? If we read the formulas as first-order logic implications, then both rules are equivalent to the disjunction $\forall x.(\neg\mathsf{human}(x) \vee \mathsf{underage}(x) \vee \mathsf{adult}(x))$, which in case of alice entails $\mathsf{underage}(\mathsf{alice}) \vee \mathsf{adult}(\mathsf{alice})$. This kind of disjunctive information cannot be captured in any single output database.

Indeed, if we extend the notion of applying a rule to negation in the natural way, the T_P operator from Section 4.2.1 oscillates on this input: $T_P^0 = \{\mathsf{human}(\mathsf{alice})\}$, $T_P^1 = \{\mathsf{human}(\mathsf{alice}), \mathsf{underage}(\mathsf{alice}), \mathsf{adult}(\mathsf{alice})\}$, $T_P^2 = T_P^0$, $T_P^3 = T_P^1$, and so on. Considering all Herbrand models as in Section 4.2.2, the only logical consequence would be human(alice), and letting ¬ appear in rules in the second-order translation of Section 4.2.4 would likewise lead to empty relations for adult and underage. For the proof-theoretic view of Section 4.2.3, it is not immediately clear at all how negation should be included, since Datalog does not offer any direct way to prove that a relation does not hold.

Finally, a practical tool that processes rules one by one (as most implementations do) might produce either $\{\mathsf{human}(\mathsf{alice}), \mathsf{underage}(\mathsf{alice})\}$ or $\{\mathsf{human}(\mathsf{alice}), \mathsf{adult}(\mathsf{alice}\}$ as its result, depending on which of the two rules is applied first.

As the previous example shows, the neatly aligned semantic views may easily diverge or cease to make sense when extending Datalog, and the result of previously correct computation procedures may depend on irrelevant details such as rule order. A major goal in designing Datalog-like rule languages has often been to prevent such breakdowns, or at least to give up as little declarativity as possible. There are exceptions, and some authors have argued that a fully operational semantics (as in "the semantics is what the algorithm does") is acceptable in exchange for the greater freedom of expression that this gives to programmers (e.g., [21]).

For negation (and also aggregation; see Section 8), there is a widely accepted well-behaved case that illustrates how a careful extension of Datalog might work. Let us first define the syntax.

▶ **Definition 29** (Datalog Rules with Negation). *A* Datalog rule with negation *has the form*

$$p(\boldsymbol{t}) \leftarrow b_1(\boldsymbol{t_1}), \ldots, b_m(\boldsymbol{t_m}), \neg n_1(\boldsymbol{s_1}), \ldots, \neg n_k(\boldsymbol{s_k}) \tag{6}$$

*where $p, b_1, \ldots, b_m, n_1 \ldots, n_k$ are predicates with the given lists of terms, such that all variables in $\boldsymbol{s_1}, \ldots, \boldsymbol{s_k}$ also occur in $\boldsymbol{t_1}, \ldots, \boldsymbol{t_m}$ (*safety*). The extension of Datalog with negation is denoted Datalog$^\neg$.*

▶ Note 30 (Negation in Datalog Systems). Actual systems have adopted different syntactic forms to express negation. Examples include `~` (Logica, Nemo), `!` (Soufflé), `NOT` (RDFox), and `not` (Clingo). Instead of requiring safety, some systems assume an alternative semantics for variables that only occur in a single negated atom. Under the usual universal quantification, an unsafe rule like $\mathsf{bachelor}(x) \leftarrow \mathsf{man}(x), \neg\mathsf{spouse}(x, y)$ would mean that bachelor(x) can be inferred if there are any values for x and y such that $\mathsf{man}(x)$ is true and $\mathsf{spouse}(x, y)$ is not, i.e., if x is a man that is not married to *all* domain elements. This is rarely wanted, and systems that allow such rules at all tend to quantify the unsafe variable existentially: $\forall x.\big(\mathsf{bachelor}(x) \leftarrow \mathsf{man}(x), \neg\exists y.\mathsf{spouse}(x, y)\big)$. This is done, e.g., in Clingo and Nemo. Where this is not supported, one can also simply split the rule into two: $\mathsf{married}(x) \leftarrow \mathsf{spouse}(x, y)$ and $\mathsf{bachelor}(x) \leftarrow \mathsf{man}(x), \neg\mathsf{married}(x)$.

Cases like Example 28 indicate that it is problematic to combine recursion with *negation as failure* (the idea that the absence of a fact may be assumed once all attempts to prove it have failed). A possible solution is to require that rules can be organised in "layers" where negation may only refer to facts that were computed in lower layers, and it is ensured that these facts are never affected by the higher rules. We can achieve that by assigning numbers to head predicates.

▶ **Definition 31** (Stratified Datalog$^\neg$). *A* stratification *of a Datalog$^\neg$ program P is a function* level *from predicate names in P to natural numbers, such that the following holds for every rule of the form* (6) *in P:*

1. $\mathsf{level}(p) \geq \mathsf{level}(b_i)$ *for all* $i \in \{1, \ldots, m\}$ *and*
2. $\mathsf{level}(p) > \mathsf{level}(n_i)$ *for all* $i \in \{1, \ldots, k\}$.

A Datalog$^\neg$ program is stratified *if it admits some stratification.*

A common way to find a stratification is to express the constraints of Definition 31 in a graph that has predicate names as vertices and edges labelled $\geq$ or $>$ based on their co-occurrence in rules. Stratification is possible if this graph has no cycle through an $>$-edge, and a suitable level mapping can in this case be obtained from a topological order of the strongly connected components of the graph. A stratification can be used to compute the output of Datalog programs by processing rules layer by layer. For the next definition, recall that $P(\mathcal{D})$ denotes the output database returned by P on input $\mathcal{D}$.

▶ **Definition 32** (Operational Stratified Datalog$^\neg$ Semantics). *Let P be a Datalog$^\neg$ program with stratification* level. *Without loss of generality, we assume that the values of* level *ranges from 0 to L, where 0 is used exactly for the EDB predicates and every $i \in \{1, \ldots, L\}$ is the level of some IDB predicate. For every $\ell \in \{1, \ldots, L\}$, let $P[\ell]$ be the set of all rules in P that have a head predicate p with $\mathsf{level}(p) = \ell$. We view $P[\ell]$ as a Datalog$^\neg$ program that has as input predicates all the input predicates of P and all IDB predicates of lower levels, and that has as output predicates all output predicates of P and all IDB predicates that occur in any rule of a higher level.*

Given an input database $\mathcal{D}^0$, we define $\mathcal{D}^i = P[i](\mathcal{D}^{i-1})$ for all $1 \leq i \leq L$. The output database of P over $\mathcal{D}$, denoted $P(\mathcal{D})$, is $\mathcal{D}^L$.

By stratification, negated atoms in $P[i]$ can only refer to predicates whose ultimate value has been computed for the intermediate database $\mathcal{D}^i$, and absence of an atom can safely be taken as a guarantee that it is really false. Implementing this semantics is not harder than implementing pure Datalog: all it needs is an iterated computation of Datalog outputs and a slightly extended check to find matching rules.

▶ Note 33 (Input Negation and Inequality). A special form of stratified negation is *input negation*, where only input predicates can occur in negated atoms. Another special case of stratified negation is inequality. Indeed, an inequality predicate $\neq$ could always be defined by adding all of the following rules (with auxiliary predicates dom and =):

$$\begin{aligned} \mathsf{dom}(x_i) &\leftarrow p(x_1, \ldots, x_{\mathsf{ar}(p)}) && \text{for all input predicates } p \text{ and all } i \in \{1, \ldots, \mathsf{ar}(p)\} \\ x = x &\leftarrow \mathsf{dom}(x) \\ x \neq y &\leftarrow \mathsf{dom}(x), \mathsf{dom}(y), \neg(x = y) \end{aligned}$$

It is not hard to see that these rules are compatible with stratification. Datalog with input negation or inequality is mainly relevant in theoretical studies of expressive power [2]. Practical systems rather support arbitrary stratified negation and provide $\neq$ as a built-in.

▶ Note 34 (Non-monotonicity). Negation (even when stratified or restricted to inputs) is the most common source of *non-monotonicity* in logic programming. Indeed, under the semantics of Definition 32, adding more facts to the input may lead to fewer conclusions – the T_P operator becomes non-monotonic. In contrast, logical entailment in first-order logic and many other logics is monotonic in the sense that additional inputs always lead to more conclusions.

The semantics of Definition 32 can also be characterised by a suitable model theory. However, we now have to restrict to a special subset of all Herbrand models, the so-called *stable models*, which are also used in answer set programming [44, 23]. It turns out that a stratified Datalog$^\neg$ program has a unique stable model – also known as the *perfect model* –, which contains exactly the output facts obtained as per Definition 32. A corresponding second-order characterisation that encodes the definition of stable model is conceivable but likely not very enlightening.

We lose the elegant proof-theoretic characterisation of Section 4.2.3. In the context of Prolog, negation as failure can be supported by so-called *SLDNF resolution*, which constructs tree-like computational schemes [4]. However, negated atoms in this case are proven by a sub-procedure that fails to find a proof for their non-negated version. It seems that such a “failure to prove” cannot be certified elegantly in a closed form. Nemo offers proof-tree like explanations for such inferences by allowing users to inspect all non-negated atoms that were derived for the predicate in question. Depending on the goal of explanation, other approaches might be possible, e.g., *counterfactual* analyses that explain how the input could have been changed to derive the missing fact, which is closely related to the concept of *abduction* [29]. Further approaches have been developed for database query results in the context of *Why-Not provenance* [15], but none seems to offer a solution that is as elegant and versatile as proof trees for pure Datalog.

7 Adding Datatypes to Datalog

Real applications often involve many concrete forms of data, such as numbers, strings, calendar dates, or geographic coordinates. The abstract Datalog framework introduced so far can already handle such dates by simply considering them as special kinds of constants, but such an *untyped* approach does not offer any means to inspect or create new data values. For example, we might want to check if a string starts with the letter A, or create a number that is the sum of two other numbers. In this section, we explain how such features are commonly added to Datalog, and what this means for a logic-based view.

7.1 Strong and Weak Typing

There are two major approaches for handling concrete data values in Datalog (and in many other languages in computer science): *strongly typed* and *weakly typed*. The strongly typed approach associates precise type information with every place where data might appear, which for Datalog is mainly in atoms. This view is typical for relational databases, where every relational table has an explicit *schema* that defines the names and types of its columns. Such schemas can readily be used with Datalog rules, and static ("compile time") checks can be performed to ensure that all rules are compatible with the stated types in the sense that no mistyped atom can be inferred. Datalog systems that follow this tradition include Logica and Soufflé.

The weakly typed approach, in contrast, associates type information with individual values, but does not restrict the types of values that may be used in a specific place. If a specific type is required to perform an operation, then this condition is checked dynamically ("at runtime"). Dynamic type conversions and graceful error handling are used to mitigate type mismatches. This view is typical in logic programming (Prolog, ASP) and fields concerned with open data exchange (knowledge graphs, web data). Datalog systems that follow this tradition include Nemo, RDFox, and Clingo.

Strong typing leads to a higher level of type safety, which can help to catch programming errors and simplify implementation. The certainty that all values that may occur at some place are of a specific type, say `int64`, is clearly very helpful to system implementers. Weak typing in turn offers greater flexibility, in particular when dealing with unknown, possibly untyped or unreliable data sources. Untyped and weakly typed data formats like CSV, JSON, or RDF dominate Web data exchange. As fundamental as the distinction often is, each approach can also attain some of the benefits of the other: strongly typed approaches can offer *union types* to support greater data variability, while weakly typed approaches can automatically deduce stronger type information by static and dynamic analyses.

▶ Note 35 (Datatypes in Datalog Systems). Whether strong or weak typing is used, the concrete types that are actually available in practical systems may also vary significantly. Traditional logic programming offers only a few types, such as string and integer, and it may even be left to implementers to select details, such as the precision of the floating point number type in Prolog [34]. Tools that are more focussed on data processing typically adopt the type system of the underlying data format, be it SQL, RDF, or JSON. Some of these systems are very simple (e.g., the only primitive types of JSON are *string*, *number*, and *boolean*), while others are very extensive (e.g., RDF builds upon the type system of XML Schema, with types like *gMonth* for Gregorian calendar months, and further allows systems to introduce derived and entirely new types).

In mathematical logic, generalisations of datatypes are called *sorts*, and the strongly typed approach corresponds to *many-sorted logics*. In addition to explicitly typed syntactic elements, such logics use a different domain of interpretation for each sort when defining models. The weakly typed approach, in contrast, up to this point is not different from the untyped abstract view of logic.

7.2 Built-in Functions and Predicates

Both sorted and unsorted logic need to be further enriched with datatype-specific predicates and functions to obtain the desired functionality. The standard approach for doing so is to introduce predicates and function symbols whose interpretation is predefined in a fixed, meaningful way, i.e., "built into" the model-theoretic semantics. For example, we might require that the ternary predicate sum contains exactly those triples $\langle a, b, c\rangle$ of natural numbers a, b, c such that $a + b = c$. Alternatively, we might define a function symbol $+$ whose interpretation is fixed to the usual mapping from pairs of natural numbers to their sum.

In practical systems, such pre-defined concrete functions and predicates are known as *built-ins*. Typical built-ins include arithmetic, string functions, and type conversion operations (e.g., to turn an integer into a string). In strongly typed systems, functions are often overloaded (e.g., + might represent natural number addition, floating point addition, or string concatenation), with type information used to select the suitable function. In weakly typed systems, a single function needs to specify all relevant cases. For example, the SPARQL query language defines how to add an integer to a float by first converting the numbers to a common type. Likewise, weakly typed languages need to specify the behaviour of comparison predicates like $\leq$ for values of different type.

Two difficulties arise when making such extensions to a logical language like Datalog:

1. Datatype-specific reasoning can be hard. When used without restrictions, even with the most common built-ins may make it very hard or impossible to find matches for applying a rule. With arithmetic functions on integers, e.g., we can define a rule body that matches exactly the integer solutions to an arithmetic equation in many variables, which are known to be uncomputable.[15] Even in cases where problems are solvable in principle (e.g., for linear equational systems over integers), finding such solutions might require sophisticated algorithms that significantly complicate the application of rules.
2. The number of inferences might be infinite. When introducing datatypes with infinite domains, such as (big) integers and strings, even simple Datalog programs can produce infinitely many inferences. Theorem 19 no longer holds if built-in functions can introduce new values during reasoning (e.g., the successor of a number). Even if we merely want to check if a particular inference is produced, this question may already be undecidable, since rules over unbounded domains can easily be used to encode universal computation models (e.g., using standard Datalog simulations of Turing machines [19]).

Challenge (1) is typically addressed in Datalog by extending the requirements for *safety* (see Note 8), such that all variables in a rule are now required to occur in a body atom that is not a built-in. Applying a rule can then be done in two steps: (1) find potential matches for the body atoms that are not built-ins, and (2) verify that all built-in conditions hold for a match. Step (2) is easy since we just need to evaluate built-ins for concrete values, rather than reasoning about possible solutions of a built-in expression. In practice, systems may relax these strict safety requirements, and consider some uses of variables in built-ins to be "safe" as well. Especially chains of equality atoms, such as $x = 42, y = x + 2$, lead to concrete bindings that can be used to narrow down the search for potential matches.

▶ Note 36 (Constraint Logic Programming and Reasoning Modulo Theories). There are logical formalisms and systems that support specific forms of datatype reasoning that go beyond the safe use of built-ins in Datalog. A notable example are *constraint logic programs* that, e.g., support unsafe use of linear inequalities over real numbers in rule bodies. A different approach is to express computational problems as *satisfiability modulo theories* (SMT), where boolean logic is combined with built-in constraints over various datatypes. Both approaches require dedicated reasoning algorithms for each specific datatype and combination of built-ins that is supported, so that adding new built-ins is significantly harder than in Datalog.

This leaves us with Challenge (2), which is closely related to termination. Indeed, if we require safety for built-in predicates, then it is easy to generalise the T_P operator of Section 4.2.1 to the new features, but the algorithm may no longer terminate after a finite number of steps. Even then, however, every fact in the (possibly infinite) T_P^∞ is inferred

[15] See `https://en.wikipedia.org/wiki/Hilbert%27s_tenth_problem`.

after finitely many steps. This yields a semi-decision procedure for checking fact entailment: iteratively compute T_P^i, stopping only if the fact of interest was entailed or no new inferences are found. This procedure will not terminate if T_P^∞ is infinite but does not contain the fact we are looking for. Nevertheless, the semantics is still arguably declarative in the sense that the outcome – even if not computable – can be mathematically defined in an implementation independent way.

Termination (or the lack thereof) can be addressed in various ways:

- Accept non-termination. Most systems do not implement any special measures and simply allow users to implement Datalog programs that do not terminate. This is sensible since it is known that automatically checking for termination is undecidable (be it for a specific input database or, more strongly, for arbitrary input databases), and meaningful programs may be very hard to analyse in this respect. Allowing for non-termination does not destroy declarativity and associated benefits (Section 3.1). Most tools, including Clingo, RDFox, Nemo, and Soufflé, fall into that category.
- Enforce termination: Some systems provide operational ways of enforcing termination through explicit stopping conditions. The (discontinued) data analysis system *Empty-Headed*, e.g., allows rules to be annotated with the specific number of times that they should be executed [1]. This is useful for iterative graph algorithms such as *PageRank*, which approximate a solution arbitrary closely without reaching a stable state. Some such termination criteria may be fully declarative, and could possibly be simulated in pure Datalog, but other approaches may lead to non-declarative behaviour where results depend on volatile conditions during execution.
- Analyse termination. Although undecidable in general, it is possible to detect at least some cases where termination or non-termination is certain. This type of analyses have mostly been proposed in relation to abstract function terms (see Section 7.3) and existential quantifiers. Cuenca Grau et al. give an overview of related termination conditions [18]. These checks are rarely seen in practice.

▶ Note 37 (Normalising terms with $=$). In the presence of an equality predicate $=$, we can introduce body atoms of the form $x = t$ to "assign" the value of a term t to a variable x. This can be used to eliminate terms t that occur anywhere in a rule: just replace t by a fresh variable x, and add $x = t$ to the body. Using this idea, we can easily transform rules so as to (1) avoid functions in rule heads, (2) avoid functions in regular (non-equality) body atoms, or (3) avoid nested function terms. In general, such transformations may not always improve readability or performance, but they are a valuable tool for simplifying definitions and implementations by reducing syntactic complexity.

7.3 Complex Value Types

Besides primitive types like integer or string, Datalog systems have also considered complex types, such as lists, maps, and sets. The primary complex type in logic programming is the type of *abstract function terms*, whose values include expressions such as $f(a, g(b))$. In contrast to built-in functions, these terms are not further evaluated but used literally. Prolog also uses nested function terms to encode lists, but many tools prefer dedicated list types.

Even without any further types, the set of nested complex values is always infinite, so that questions of termination and complexity become relevant. Datalog has originally been derived from Prolog, essentially by removing abstract functions and negation. Yet function symbols are supported by answer set programming engines (e.g., Clingo). Soufflé's *Algebraic Data Types* and Logica's *records* support similar structures. Strongly types frameworks like

Soufflé declare strict schema information also for abstract functions, whereas weakly typed frameworks like Clingo merely consider an arity for each function symbol.

Lists and sets are the main kinds of *complex values* studied for Datalog [2], and recent works have also clarified decidability and complexity results for fragments of that language [51]. Implementations of complex values in Datalog still remain limited today. An inactive branch of the ASP engine DLV supports lists and sets natively [11], Logica supports lists (called *arrays*), and the N3 rule language draft has a list datatype [68]. The small-scale Datalog tool Percival supports lists and maps based on JSON [73]. Support for lists, maps, and function terms in Nemo is planned.[16]

8 Aggregation

In databases, aggregation is a computation that maps groups of tuples (database rows) to single values. The simplest example is `COUNT`, which counts the tuples per group. Other common aggregates include `SUM`, `MIN`, and `MAX`. Aggregation functions are supported in many Datalog engines, but some additional complications need to be taken care of.

Let us consider a weakly typed setting with a single domain Δ of database elements. To apply aggregation on a relational database model, the following details must be specified:

- an input relation (set of tuples) $R \subseteq \Delta^n$ of arity n,
- two disjoint lists of positions, G (“group by”) and A (“aggregate”), from the set $\{1, \dots, n\}$, and
- an aggregation function $f : 2^{\Delta^{|A|}} \to \Delta$, which maps from $|A|$-ary relations to individual domain elements.

In practice, aggregate functions are often defined over arbitrary arities $|A|$. This may be obvious (e.g., for `COUNT`) or it may require some additional specification (e.g., `SUM` might sum up all values in all tuples or only the values in the first column). The result of aggregation is then defined as follows:

▶ **Definition 38** (Aggregation over a relation)**.** *Consider relation R, position lists G and A, and aggregation function f as above. For a tuple $\boldsymbol{t} = \langle t_1, \dots, t_n \rangle \in R$, we introduce the notation $G(\boldsymbol{t}) := \langle t_{G[1]}, \dots, t_{G[\mathsf{len}(G)]} \rangle$ for the tuple obtained by projecting to the positions in G. The projection $A(\boldsymbol{t})$ is defined analogously.*

Now the aggregation of R w.r.t. G, A, and f is defined through the following steps:

1. ***Grouping:*** *Let $R_G := \{G(\boldsymbol{t}) \mid \boldsymbol{t} \in R\}$ and $R_A := \{A(\boldsymbol{t}) \mid \boldsymbol{t} \in R\}$.*
   The function $g : R_G \to 2^{R_A}$ is defined by setting $g(\boldsymbol{s}) := \{A(\boldsymbol{t}) \mid \boldsymbol{t} \in R, G(\boldsymbol{t}) = \boldsymbol{s}\}$.
2. ***Apply aggregation:*** *The aggregated relation is $\{\langle \boldsymbol{s}, f(g(\boldsymbol{s}))\rangle \mid \boldsymbol{s} \in R_G\}$.*

Therefore, the grouping function g assigns a set of A-tuples to every G-tuple, and this set is then aggregated into a single domain value to obtain output tuples. Note that we do not require that all positions of R occur in G or A: positions that do not occur, would simply be projected away. This is particularly significant since Definition 38 is based on sets, so that grouped tuples $g(\boldsymbol{s})$ may not contain duplicates. We further discuss this aspect in the next section with the help of concrete examples.

[16] Progress is tracked at `https://github.com/knowsys/nemo/issues/699`.

8.1 Syntax and Semantics of Individual Aggregates

To integrate aggregation into Datalog, we have to specify R, G, A, and f of Definition 38 within rules. There is no generally accepted notation for aggregation in logic, so we start by explaining the syntax used in Nemo. Other systems have proposed different syntactic forms, and sometimes also (subtly) different semantics, as we will discuss below.

▶ **Example 39.** Nemo places aggregation in rule heads, following the intuition that aggregation happens on the matches of a rule to compute a value used in the head. Suppose we have data on annual company-specific greenhouse gas emissions, given in facts such as $\mathsf{emission}(\mathsf{ExxonMobil}, \mathsf{US}, 2023, 562.224)$, stating that ExxonMobil is a US company that in the year 2023 caused emissions equivalent to over 562 million tons of CO_2. For a basic example, we wish to find out for how many countries there is data in each year:

```
countriesPerYear(?year, #count(?country)) :- emission(?cmp, ?country, ?year, ?amount) .
```

The symbol # indicates an aggregate function. The rule is processed as follows:

1. Find all matches for the rule body.
   Result: a relation of arity 4 that agrees with the input relation for emission.
2. Group tuples that agree on all non-aggregated head variables.
   Result: one group per year, containing a set of all tuples for that year.
3. Project the grouped tuples so that only the variables mentioned in the aggregate remain.
   Result: one group per year, containing a set of all countries for that year.
4. Apply aggregate function to groups and store inferred fact.
   Result: facts with one number per year.

In the notation of Definition 38, the input relation R can therefore be taken to be the set of all tuples of values that match the rule body (in some fixed but arbitrary order, making variables correspond to positions). The group-by positions G are defined by the variables in the head that appear outside of aggregates, and the aggregate positions A are defined by the variables occurring in the aggregate function (in the given order). This definition assumes that all non-aggregate terms in the head are variables: other terms can be replaced by fresh variables as explained in Note 37.

An important issue for aggregation in Datalog is that we generally consider *sets* of tuples, which cannot contain duplicates. Hence, in Example 39, each country was counted only once, even for years where several companies from that country had data. This is different from relational databases, where duplicates are kept by default and the keyword `DISTINCT` is required to remove them.

▶ **Example 40.** Continuing Example 39, we wish to compute the sum of all emissions caused by each company. Nemo has an aggregate `#sum`, so we might write:

```
total(?company, #sum(?amount)) :- emission(?company, ?country, ?year, ?amount) .
```

However, this would not be correct: step (3) in Example 39 would project the aggregation tuples to become a unary relation that specifies only the emission amounts. Therefore, if the same amount was emitted in two distinct years, then this duplicate would be removed, and the amount would feature only once in the sum. To prevent this, we need to leave distinguishing information in the aggregation tuples:

```
total(?company, #sum(?amount, ?year)) :- emission(?company, ?country, ?year, ?amount) .
```

The aggregate now runs on a binary relation that may contain the same amount several times with different years. The function `#sum` is defined to add up the values in the first parameter, so the years are not added (if we wanted this, we could write `#sum(?amount + ?year)`). Nemo therefore can use set semantics throughout while still offering full control over (de)duplication of tuples.

Other Datalog systems make different choices on how to specify aggregation and the necessary details (group-by variables and duplicate elimination policy).

▶ **Example 41.** Aggregate expressions in Soufflé are sub-expressions with own "bodies", but group-by variables must be bound *outside* of these expressions, and duplicate elimination (projection) seems to require auxiliary rules. Example 39 could be expressed as:[17]

```
auxYearCountry(year, country) :- emission(_, country, year, _) .
    countriesPerYear(year, c) :- auxYearCountry(year, _),
                                 c = count : { auxYearCountry(year, _) } .
```

Clingo uses syntactically similar aggregate expressions but, like Nemo, allows specifying aggregation variables and eagerly removes duplicates before aggregation:

```
countriesPerYear(YEAR, C) :- emission(_,_,_,YEAR),
                             C = #count{ COUNTRY : emission(_,COUNTRY,YEAR,_)} .
```

RDFox also uses nested aggregate expressions, but specifies group-by variables and aggregate variables, and may return bindings for multiple variables. RDFox also can control duplicate elimination, so that, besides the nesting, its aggregate expressions are functionally equivalent to Nemo aggregate rules.

These examples underline that aggregation in logic programming is much less standardised than other features. Most uses of aggregation can still be expressed in all systems, possibly with some auxiliary rules. A special case that is not treated uniformly are "empty groups". If aggregation follows Definition 38, then there can never be empty groups: if a group would be empty, then it does not occur in R_G at all. This is also the case for Nemo, whereas Soufflé and Clingo define aggregation based on two input relations: one for specifying groups (outside of the nested aggregate), and one for specifying the tuples to be aggregated per group (inside the nested aggregate). Therefore, the `#count` function in Nemo will never return 0, whereas Soufflé's nested `count` with externally defined group-by variables might. In Nemo, the same result can be achieved by specifying an additional rule that uses negation to derive value 0 for empty groups. Both approaches are therefore equivalent in terms of expressive power. Systems that enable aggregation over empty groups must specify the value of all aggregation functions for $\emptyset$, which is not always as intuitive as for `count`, e.g., when considering aggregates such as `average` or `min`.

8.2 Aggregates in Datalog Programs

Now that we know how aggregation might be used in single rules, we should ask how this feature interacts with the semantics of whole Datalog programs. Since aggregate rules refer

[17] Unrelated to aggregation, this example uses underscores _ to denote anonymous variables, a common notation supported in most Datalog engines (including Clingo, Nemo, RDFox, and Soufflé). We could equivalently replace each occurrence of _ by a variable that is not used anywhere else.

to a complete set of facts, rules can no longer be applied to individual matches but ideally should be computed when all matches are known. This can be ensured through the concept of *stratification* as introduced for negation in Definition 31. For the following definition, we assume Nemo-style aggregation in rule heads as in Example 39, but analogous definitions can also be given for other syntactic variants [55].

▶ **Definition 42** (Stratified Datalog$^\neg$ with Aggregates). *A* stratification *of a Datalog$^\neg$ program P with aggregates is a function* level *from predicates in P to natural numbers, such that:*
1. *for every aggregation-free rule in P,* level *satisfies the conditions of Definition 31,*
2. *for every rule in P that uses aggregation, the* level *of head predicates is strictly greater than the level of any body predicate.*

A Datalog$^\neg$ program with aggregation is stratified *if it admits some stratification.*

With such strict requirements in place, aggregation rules can always be safely executed exactly once during the computation of T_P. Datalog systems that currently require aggregations to be stratified include Nemo, RDFox, and Soufflé. Similarly to unstratified negation (see Example 28), unstratified aggregation can lead to situations where no unique result can be determined. While not desirable in Datalog, this is the normal situation in Answer Set Programming, and ASP engines such as Clingo do indeed allow for unstratified aggregation (the specific semantics is beyond the scope of this paper).

The underlying challenge is that aggregation often introduces non-monotonic behaviour, where adding facts to the input may cause a previously inferred fact to be no longer valid. Nevertheless, there are cases where aggregation could meaningfully be used in recursive rules without giving up on the uniqueness of the overall output.

▶ **Example 43.** Consider a weighted directed graph where each edge is given as a fact edge(source, target, cost). The following non-stratified Nemo program might be a suggestive attempt of computing the shortest path between any pair of nodes:

```
72           path(?s,?t,?c) :- edge(?s,?t,?c), ?c>0 .
73 path(?s,?t,#min(?cp+?c)) :- path(?s,?m,?cp), edge(?m,?t,?c), ?c>0 .
```

Both rules restrict costs to be positive, as considering negative costs might make the solution undefined if the graph has cycles of negative total cost. The rule in L72 is a simple initialisation. The rule in L73 uses a `#min` aggregate that computes the least value in a group, which in this case is the least cost of any path from `?s` to `?t` that can be found by extending known paths. This use of aggregation is highly recursive and non-monotonic. To illustrate, we consider the following five edges:

```
74 edge(s,b,2) .   edge(b,t,2) .   edge(s,c,1) .   edge(c,d,1) .   edge(d,t,1) .
```

The path $s \to b \to t$ has a higher cost than the longer path $s \to c \to d \to t$. If we compute T_P as before (showing only paths for source s), we first obtain $\mathsf{path}(s, b, 2), \mathsf{path}(s, c, 1) \in T_P^1$. In the next step, L73 is yields two inferences $\mathsf{path}(s, t, 4), \mathsf{path}(s, d, 2) \in T_P^2$, each obtained by aggregating a one-element group. In the following step, however, L73 yields $\mathsf{path}(s, t, 3) \in T_P^2$ as an aggregate of two body matches: $\mathsf{path}(s, b, 2), \mathsf{edge}(b, t, 2)$ (the one from before) and $\mathsf{path}(s, d, 2), \mathsf{edge}(d, t, 1)$ (new). At this point, there is no longer any justification for having derived $\mathsf{path}(s, t, 4)$, since it follows from none of the rules.

Nevertheless, this particular program features some kind of monotonicity as well: adding more facts and computing more paths will always make the total costs smaller. Indeed, an algorithm that simply updates the least cost would reliably (and efficiently) compute all shortest paths. It is possible to capture the same result declaratively in stratified Datalog:

```
             path(?s,?t,?c) :- edge(?s,?t,?c), ?c>0 .
          path(?s,?t,?cp+?c) :- path(?s,?m,?cp), edge(?m,?t,?c), ?c>0 .
shortestPath(?s,?t,#min(?c)) :- path(?s,?t,?c) .
```

Here, we first compute all path costs recursively, and then, in a higher level of the stratification, select the minimal cost for each source-target pair. While this captures the intended semantics, mathematically speaking, it is not a feasible solution in practice, since cycles in the graph lead to arbitrarily long and costly paths. The iterative evaluation of the first two rules would therefore not terminate.

The potential value of a non-stratified use of aggregates has been recognised long ago. In the wider field of logic programming (where, e.g., ASP is included), many different cases and related semantics have been studied (Liu and Stoller give a recent overview [45]). In Datalog more specifically, aggregates with a *monotonic* behaviour have gathered most interest [60], but it was also observed that it is difficult to recognise this favourable behaviour in general Datalog rules with aggregation [67]. Various language extensions have been proposed to add non-stratified aggregation in restricted ways that would be easier to detect, including the use of *monotonic aggregates* [63], *limit parameters* that correspond to functional monotonic min and max [37], custom semantics based on ordered algebraic structures [47, 38], and syntactic criteria for important special cases [72]. As of today, very few tools seem to have developed such functionality beyond prototype stage, notably the Datalog-influenced programming language Flix [47]. SocialLite also includes special support for non-stratified aggregation [62], but the system has not been updated since 2016.

9 Hands-On: Enriching and Analysing Data

Using our knowledge of negation and aggregation, we can turn the analysis of CO_2 emissions from Example 39 into a program over real data. Suitable emission figures are researched and published by Carbon Majors (`https://carbonmajors.org/Downloads`) in the text-based CSV format. Unfortunately, this data contains merely the names of companies (and other organisations), but does not specify which country they have their headquarters in. To get this information, we need to integrate the data with content from Wikidata.

The hands-on therefore illustrates two further capabilities of Datalog: data integration and distributed query answering. Data integration is difficult since it is hard to make connections between elements of different data sources. For example, the Carbon Majors refers to an organisation named *Chevron* of type *Investor-owned Company*, but Wikidata knows eleven entities of that name, including a painting by Peter Wells and a genus of moss-dwelling animals, but no company. However, there are over 1000 entities that contain *Chevron* in their name, including companies, such as Chevron Corporation, Chevron U.S.A Inc., and Chevron Engineering Ltd. Obviously, it can be hard or impossible to match such elements in a fully automated way.

This is a known challenge, and unique *identifiers* have been established in many domains to refer to entities unambiguously. Well known IDs include ISBN numbers for books and ORCIDs for academic researchers. Overall, Wikidata knows over 9,500 distinct identifiers that items can be connected to. For companies and other legal organisations, the *Legal Entity Identifier* (LEI) is relevant, and this identifier is contained in the Carbon Majors data in many cases, but not in all.

Our data processing strategy therefore is as follows. First, we look for Wikidata entities with a LEI as specified by Carbon Majors. Second, for the cases where no match was found,

```
@prefix wdqs: <https://query.wikidata.org/> .

@parameter $yearOfInterest = 2023 .
@parameter  $wdSELECT = """
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX bd: <http://www.bigdata.com/rdf#>
SELECT """ .

% Carbon Majors data. Schema: cmData(year,organisation,org-type,LEI,Mt CO2 equiv.)
@import cmData :- csv{
  resource=CONCAT("https://raw.githubusercontent.com/knowsys/nemo-examples/main/",
                  "examples/carbon-emissions/emissions_low_granularity.csv"),
  ignore_headers=true, % first line is CSV header here
  format=(int,string,string,string,double)
} .
% Find Legal Entity Id on Wikidata. Schema: wdLegalEntityId(LEI,QID)
@import wdLegalEntityId :- sparql{
  endpoint=wdqs:sparql,
  query=CONCAT($wdSELECT,"""?lei ?qid WHERE { ?qid wdt:P1278 ?lei . }""")
} .
% Find Wikidata businesses by English name. Schema: wdBusiness(QID,label)
@import wdBusiness :- sparql{
  endpoint=wdqs:sparql,
  query=CONCAT($wdSELECT,"""?qid ?langLabel WHERE {
    ?qid rdfs:label ?langLabel; wdt:P31/wdt:P279* wd:Q4830453 . }""")
} .
% Find Wikidata country information for ID. Schema: wdCountry(QID,countryQID)
@import wdCountry :- sparql{
  endpoint=wdqs:sparql,
  query=CONCAT($wdSELECT,"""?qid ?countryId WHERE { ?qid wdt:P17 ?countryId }""")
} .
% Find Wikidata label for any ID. Schema: wdLabel(QID,label)
@import wdLabel :- sparql{
  endpoint=wdqs:sparql,
  query=CONCAT($wdSELECT,"""?qid ?qidLabel
    WHERE { SERVICE wikibase:label { bd:serviceParam wikibase:language "en" } }""")
} .

% 1. try to find organisation on Wikidata by its LEI identifier:
orgIdFromLEI(?org,?id) :- cmData(_,?org,_,?lei,_), ?lei!="", wdLegalEntityId(?lei,?id) .
% 2. try to find organisation on Wikidata by its name:
   businessId(?org,?id) :- cmData(_,?org,_,?lei,_), ~orgIdFromLEI(?org,_),
                           wdBusiness(?id,STRLANG(?org,"en")) .
    ambiguousName(?org) :- businessId(?org,?id1), businessId(?org,?id2), ?id1!=?id2 .
orgIdFromName(?org,?id) :- businessId(?org,?id), ~ambiguousName(?org) .
% 3. fetch country information for every organisation found on Wikidata:
orgCountry(?org,?countryId) :- orgIdFromLEI(?org,?qid), wdCountry(?qid,?countryId) .
orgCountry(?org,?countryId) :- orgIdFromName(?org,?qid), wdCountry(?qid,?countryId) .
% 4. compute emissions per country (label) & sum up the rest under "Missing country":
countryEmissionsPerYear(#sum(?Mtco2eq),?country) :-
    cmData($yearOfInterest,?org,?type,?lei,?Mtco2eq), orgCountry(?org,?countryId),
    wdLabel(?countryId,?countryLabel), ?country=STR(?countryLabel) .
countryEmissionsPerYear(#sum(?Mtco2eq),"Missing country") :-
    cmData($yearOfInterest,?org,?type,?lei,?Mtco2eq), ~orgCountry(?org,_) .

@output countryEmissionsPerYear .
```

Listing 3 Nemo Datalog program to compute how pollution by the world-largest carbon emitting organisations distribute over countries in the year 2023

we match organisation names with the names of businesses in Wikidata (only exact matches). To increase confidence, we only use matches that are unique. Third, we query Wikidata for country information about the matched organisations, and finally we compute the aggregated emission values per country. These four stages form natural levels in a stratification.

The complete Nemo program can be seen in Listing 3 and at `https://tud.link/r44q79`. We aggregate the data for year 2023 (L80). To fit a single page, some (otherwise unnecessary) abbreviations are used. In particular, the string concatenation function `CONCAT` is used in many places. A longer string that we use in all SPARQL queries is defined in L81.

The relevant data sources are declared in the imports on L90–L117. A new feature here is the CSV import (L90), which fetches an online file.[18] CSV is a string-based format that does not include type information, hence we specify how each column should be interpreted (L94); this also informs Nemo about the arity (5) of the imported predicate `cmData`.

The SPARQL imports use similar queries as in the hands-on in Section 5. Relevant new Wikidata properties are P17 (country) and P1278 (Legal Entity Identifier). The query for `wdBusiness`, which we use to find business by their name, includes some extra conditions to restrict to items classified as business (Q4830453) on Wikidata.

The rules in L120–L134 implement the processing stages outlined above. They should mostly be easy to understand, but there are some features that we have not explained before. First, some of the rules are using Nemo's negation operator, written as `~`. Second, several rules use *anonymous variables*, indicated by an underscore `_`. This common abbreviation from logic programming is used to indicate distinct variables that occur only once in a rule, so each underscore denotes a different variable. Finally, we use functions `STRLANG` (L123) and `STR` (L132). These are related to a new type of data that we have not encountered yet: the *language-tagged string*. These are pairs of string and language identifiers, such as `"Germany"@en` or `"Allemagne"@fr`. All labels in Wikidata have this form, so in L123 we use `STRLANG` to construct a language-tagged string (with tag `en`) to query `wdBusiness`. Conversely, we use `STR` to extract plain strings (this function can also be used to turn other values into string representations). It is very common in data integration that datatypes do not match and need to be converted with suitable functions.

As a final remark, note how we use negation and aggregation in rule L133 to sum up all emissions from organisations for which no country could be found. This is an important measure to see how much has still been missed, so as to avoid drawing premature conclusions from the remaining outputs. In general, Datalog makes it convenient to compute details about unexpected or mismatched cases. In data analysis, errors and mismatches can occur easily, and it is often difficult to judge if final results are correct or not. Diligent analysts will therefore double check all outputs, e.g., by computing additional views.

As an exercise to the reader, the program in Listing 3 can be modified and improved in several ways. One option is to further weigh annual emissions by the country's size, measured in terms of its inhabitants (the Wikidata property for population is P1082). Another option is to compute and analyse cases where the matching with Wikidata has failed, and to think of ways of covering these cases. With very little information to go by on the CSV side, it might be necessary to consider manual mappings for some of the remaining organisations. Since Wikidata, like Wikipedia, can be edited, it is also possible to complete or correct LEI

[18] The URL from which we import here holds an unmodified copy of the file found at `https://carbonmajors.org/evoke/391/get_cm_file?type=Basic&file=emissions_low_granularity.csv`. Using the original URL instead only works in Nemo command line, since most browsers will block Nemo Web from loading this file due to a server misconfiguration at Carbon Majors.

information there to improve the matching.

10 Generalised Truth Values

Facts in Datalog might assume two truth values – true or false –, but many other systems of truth values have been studied. For example, facts in fuzzy logic may have a fuzzy *degree* of truth, expressed as a real number between 0 (“false”) and 1 (“true”). In databases, the influential work of Green et al. popularised the idea that database query results, too, might be qualified by a “provenance” value [30]. Such provenances might express degrees of truth (confidence, probability, degree of membership, etc.), but also many other concepts such as quantity (how often a fact was obtained), access restrictions (who is allowed to see the fact), explanations (how was a fact was derived).

The key to such approaches is to generalise not just the possible truth values but also the boolean algebra that is used to combine values. Green et al. proposed to focus on generalisations of conjunction and disjunction, with the resulting algebraic structure being that of a *commutative semiring*.[19] The same two operations are also required for generalising Datalog. Conjunctions are obviously needed to compute the truth value of rule bodies: every match of a rule may now receive a certain value, computed by combining the values of its individual atoms. Disjunctions come into play when a fact is inferred in multiple ways: the values of each inference are disjunctively combined to obtain a single overall value. Therefore, when applying rules in such a setting, we are not merely adding new inferences, but also updating the values of previous inferences. Green et al. propose a corresponding generalisation of Datalog, which additionally requires that the underlying semiring supports fixed points to which iterated computations can converge [30].

This leads to a very powerful generalisation of Datalog, and some research has been conducted to determine which algebraic properties are relevant to obtain a meaningful semantics and efficient algorithms. However, as of 2025, systems based on this idea seem to be elusive. Khamis et al. propose Datalog$^\circ$, which is based on certain pre-semirings, and report practical experiments on “an unreleased commercial system” [38]. The most advanced actually available implementation of a similar principle seems to be *Flix*, which introduces programmer-defined *lattice* structures that act as a (special type of) semiring [47]. Lattice values in Flix are syntactically represented as special parameters, but their built-in semantics is similar to truth-value generalisations.

Other generalisations of truth values can also be found in many works that are not based on semiring provenance, e.g., related to reasoning with probabilities or validity times. A notable practical systems of that type is *ProbLog*, which can be used as a reasoner for Datalog over facts that are “weighted” by probabilities [25]. Approaches for temporal and stream reasoning often annotate facts with sets of time points (or intervals) where they are valid, with examples including LARS [5] and DatalogMTL [70], for which a Python-based reasoner has been presented recently [71].

▶ Note 44 (Generalised Truth Values as Data). We have seen in Section 7 that relations in Datalog may contain many types of data, including numbers and times. Instead of using a non-standard logic with generalised truth values, we might therefore consider to manage such

[19] A commutative semiring is an algebraic structure $(K, \oplus, \otimes, 0, 1)$ over a set K two binary operations $\oplus$ and $\otimes$ on K and two distinguished elements $0, 1 \in K$ (the neutral elements for $\oplus$ and $\otimes$, respectively). One further requires that $\oplus$ and $\otimes$ are commutative, that $\otimes$ distributes over $\oplus$, and that all products with 0 yield 0 [30].

values as regular data. For example, the claim that the chance of rain in Dresden is 10% on 12 September 2025 could be expressed by a probabilistic fact rain(dresden, 2025-09-12) with probability value 0.1, but also by a regular fact rain(dresden, 2025-09-12, 0.1). Whether this is practical depends on our specific modelling requirements, since the Datalog program should also capture the intended semantics of such values. In the case of probabilities, we can capture some simple mechanics with arithmetic functions:

```
rainstorm(?place,?time,?p1*?p2) :- rain(?place,?time,?p1), storm(?place,?time,?p2) .
```

However, it will be difficult or impossible to formalise advanced aspects of probability theory in this manual way, and using ProbLog might indeed be a better choice.

Nevertheless, not all domains are as complex as probability theory, and many relevant calculations can readily be done inside Datalog without requiring a special logical theory. For example, reasoning in some fragments of LARS (where facts are annotated with validity times) can be translated into plain Datalog with time information stored in additional parameters [66], and provenance annotations (corresponding to sets of sets of inference rule applications) can be presented by auxiliary elements that are added to each fact [22]. A benefit of such approaches is that standard reasoners can be used, but also that multiple types of data can be handled in one system. Indeed, our weather example might actually require probabilistic and temporal reasoning, and it will be very difficult to find a system that supports this specific type of generalised truth value.

▶ Note 45 (Fake Generalised Truth Values). Some systems support statements that *look* like generalised truth values but are merely syntactic sugar for plain Datalog. For example, Logica lets us state a fact `rain(dresden, 2025-09-12) = 0.1`; the rule of Note 44 could then be written (using Logica syntax) as

```
rainstorm(place,time,p1*p2) :- p1 == rain(place,time), p2 == storm(place,time) ;
```

However, this is just syntactic sugar for the same standard Datalog model as in Note 44, i.e., the predicates rain and storm both have three parameters, and their use as function terms is merely a readable way to access the "special" third parameter. Similar syntax is also supported in Yedalog [16] and Dynalog [21]. Since these systems also allow us to declare unsafe rules, we can make definitions like `square(x) = x*x`; this can lead to very elegant programs that use some predicates like user-defined functions.

11 Summary and Further Reading

This chapter has given an overview of Datalog and some of its most common functional extensions, including stratified negation, datatypes, aggregation, and generalised truth values. All of these extensions have in common that major mathematical principles and fundamental usage patterns remain the same – the result is still a "Datalog-like language". This is in contrast with many other languages that also contain Datalog as a special case, and could therefore rightfully be called extensions of Datalog. Answer Set Programming (ASP), e.g., generalises Datalog, too, but the application areas and ways in which ASP is used in practice are significantly different [23]. Following the definition of the semantics in Section 4.2.2, first-order logic entailment reasoning could also be regarded as a generalisation of Datalog query answering, and theorem provers could be applied to this language. Again, however, the target applications and usage patterns are completely different from those of Datalog systems.

Nevertheless, there are also Datalog-like feature extensions that have not been discussed here. One that has been intensively researched is the extension of Datalog rules with existentially quantified variables in rule heads, leading to what is called *existential rules* or *tuple-generating dependencies*. Datalog's operational semantics (T_P operator, Section 4.2.1) can be extended to this setting by creating new placeholder elements to satisfy existential restrictions [20, 50, 41]. Motivations for existential rules stem from the theory of data integration and ontology-based query answering [54]. Nemo supports existentially quantified variables in heads (distinguished by exclamation marks, as in `!X`).

Another extension in some Datalog engines is support for abstract relations with special built-in semantics. Most prominent are equivalence relations, which are relevant in several applications. In general, one can always *axiomatise* the properties of equivalence (reflexivity, symmetry, transitivity) with Datalog rules, but this is not efficient in comparison to using data structures specifically designed for storing equivalence classes. Soufflé therefore allows binary relations to be declared equivalences, and RDFox supports special handling of a built-in equality (different from identity). Dedicated treatment can also be given to other special types of relations, but this is closer to an optimisation technique than to an actual language extension.

Some other topics have deliberately been omitted from this chapter. One is the notable history of Datalog, from early trends in databases and expert systems, to the renaissance in the wake of ontological reasoning and declarative data analysis. An extended review is given by Maier et al. [48]. Another omission are classical and recent reasoning and implementation approaches. Decades-old methods like *semi-naive evaluation* and *magic sets* [2] are still relevant, but Datalog systems have also taken advantage of more recent developments, such as column-based database layouts [65], compiler-optimised reasoning pipelines [36], and worst-case optimal joins [35]. Algorithmic advances have been made, e.g., in the context of efficient incremental updates [53].

Besides such practical questions, there is also relevant foundational work in relation to Datalog. A classical topic is expressive power, i.e., the question which queries can be expressed at all (without considering efficiency). Datalog programs characterise polynomial functions from input to output database (see Theorem 19), but not all such functions can be expressed in Datalog. To truly *capture* PTime, Datalog can be extended with input negation and a totally ordered database domain [2]. Recent works have also shed light on the expressivity with respect to monotonic (homomorphism-closed) PTime queries [61]. Existential rules, on the other hand, can express any Turing recognisable query, and the fragment for which the standard bottom-up algorithm is guaranteed to terminate still captured the decidable queries [8]. Another foundational topic is the semantic analysis of Datalog programs. For example, it is undecidable if a Datalog program is equivalent to one without recursion (a property known as boundedness) [33]. Likewise, equivalence and containment of Datalog programs is undecidable, but several broad fragments have been identified where decidability can be recovered, albeit at very high complexities [9].

After all, the broad relevance and popularity of Datalog is making it impossible to give a comprehensive overview of even the most important works and ideas in the area. Nevertheless, this text has hopefully succeeded in conveying an impression of the distinctive modelling and usage patterns that have emerged for modern Datalog applications. Many meaningful advances are still to be made in research and in practice. Contributions are welcome.

References

1 Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017. `doi:10.1145/3129246`.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley, 1994.

3 Christian Alrabbaa, Stefan Borgwardt, Philipp Herrmann, and Markus Krötzsch. The shape of $\mathcal{EL}$ proofs: A tale of three calculi. In *Proc. 38th Int. Workshop on Description Logics (DL'25)*, CEUR Workshop Proceedings. CEUR-WS.org, 2025. To appear, preprint at `https://doi.org/10.48550/arXiv.2507.21851`.

4 Krzysztof R. Apt and Maarten H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982. `doi:10.1145/322326.322339`.

5 Harald Beck, Minh Dao-Tran, and Thomas Eiter. LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell.*, 261:16–70, 2018. `doi:10.1016/J.ARTINT.2018.04.003`.

6 Brandon Bennett. *Logical representations for automated reasoning about spatial relationships*. PhD thesis, University of Leeds, UK, 1997. URL: `http://etheses.whiterose.ac.uk/1271/`.

7 Véronique Benzaken, Evelyne Contejean, and Stefania Dumbrava. Certifying standard and stratified datalog inference engines in ssreflect. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Proc. 8th Int. Conf. Interactive Theorem Proving (ITP'17)*, volume 10499 of *LNCS*, pages 171–188. Springer, 2017. `doi:10.1007/978-3-319-66107-0_12`.

8 Camille Bourgaux, David Carral, Markus Krötzsch, Sebastian Rudolph, and Michaël Thomazo. Capturing homomorphism-closed decidable queries with existential rules. In Diego Calvanese, Esra Erdem, and Michael Thielscher, editors, *Proc. 18th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'21)*, pages 141–150. IJCAI, 2021. `doi:10.24963/KR.2021/14`.

9 Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. Reasonable highly expressive query languages. In Qiang Yang and Michael Wooldridge, editors, *Proc. 24th Int. Joint Conf. on Artificial Intelligence (IJCAI'15)*, pages 2826–2832. AAAI Press, 2015.

10 Martin Bromberger, Irina Dragoste, Rasha Faqeh, Christof Fetzer, Larry González, Markus Krötzsch, Maximilian Marx, Harish K. Murali, and Christoph Weidenbach. A sorted Datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In Dana Fisman and Grigore Rosu, editors, *Proc. 28th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'22)*, volume 13243 of *LNCS*, pages 480–501. Springer, 2022. `doi:10.1007/978-3-030-99524-9_27`.

11 Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. An ASP system with functions, lists, and sets. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Proc. 10th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 483–489. Springer, 2009.

12 Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *Theory Pract. Log. Program.*, 20(2):294–309, 2020.

13 David Carral, Irina Dragoste, and Markus Krötzsch. Reasoner = logical calculus + rule engine. *Künstliche Intell.*, 34(4):453–463, 2020. `doi:10.1007/S13218-020-00667-6`.

14 Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer, 1990.

15 Adriane Chapman and H. V. Jagadish. Why not? In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proc. 2009 ACM SIGMOD Int. Conf. on Management of Data*, pages 523–534. ACM, 2009. `doi:10.1145/1559845.1559901`.

16 Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Josef Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring knowledge at scale. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *Proc. 1st Summit on Advances in Programming Languages (SNAPL'15)*, volume 32 of *LIPIcs*, pages 63–78. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPICS.SNAPL.2015.63.

17 Cognitect, Inc. *Datomic Documentation: Rules*, retrieved 04 Sept 2025. https://docs.datomic.com/query/query-data-reference.html#rules.

18 Bernardo Cuenca Grau, Ian Horrocks, Markus Krötzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. Acyclicity notions for existential rules and their application to query answering in ontologies. *J. of Artificial Intelligence Research*, 47:741–808, 2013.

19 Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

20 Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In Maurizio Lenzerini and Domenico Lembo, editors, *Proc. 27th Symp. on Principles of Database Systems (PODS'08)*, pages 149–158. ACM, 2008.

21 Jason Eisner and Nathaniel Wesley Filardo. Dyna: Extending datalog for modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Proc. 1st Int. Datalog Reloaded Workshop, Revised Selected Papers*, volume 6702 of *LNCS*, pages 181–220. Springer, 2010. doi:10.1007/978-3-642-24206-9_11.

22 Ali Elhalawati, Markus Krötzsch, and Stephan Mennicke. An existential rule framework for computing why-provenance on-demand for datalog. In Guido Governatori and Anni-Yasmin Turhan, editors, *Proc. 2nd Int. Joint Conf. on Rules and Reasoning (RuleML+RR'22)*, volume 13752 of *LNCS*, pages 146–163. Springer, 2022.

23 Wolfgang Faber. An introduction to answer set programming and some of its extensions. In Marco Manna and Andreas Pieris, editors, *Proc. 16th Int. Reasoning Web Summer School (RW'20)*, volume 12258 of *LNCS*, pages 149–185. Springer, 2020. doi:10.1007/978-3-030-60067-9_6.

24 Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

25 Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory Pract. Log. Program.*, 15(3):358–401, 2015. doi:10.1017/S1471068414000076.

26 Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, Christian Schallhart, and Cheng Wang. DIADEM: Thousands of websites to a single database. *Proc. VLDB Endow.*, 7(14):1845–1856, 2014. doi:10.14778/2733085.2733091.

27 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019.

28 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012. doi:10.1016/j.artint.2012.04.001.

29 Georg Gottlob, Reinhard Pichler, and Fang Wei. Efficient Datalog abduction through bounded treewidth. In *Proc. 22nd AAAI Conf. on Artificial Intelligence (AAAI'07)*, pages 1626–1631. AAAI Press.

30 Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In Leonid Libkin, editor, *Proc. 26th Symp. on Principles of Database Systems (PODS'07)*, pages 31–40. ACM, 2007. doi:10.1145/1265530.1265535.

31 Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *CodeQuest:* scalable source code queries with Datalog. In Dave Thomas, editor, *Proc. 20th European Conf. Object-Oriented Programming (ECOOP'06)*, volume 4067 of *LNCS*, pages 2–27. Springer, 2006. doi:10.1007/11785477_2.

32 Steve Harris and Andy Seaborne, editors. *SPARQL 1.1 Query Language.* W3C Recommendation, 21 March 2013. Available at http://www.w3.org/TR/sparql11-query/.

33 Gerd G. Hillebrand, Paris C. Kanellakis, Harry G. Mairson, and Moshe Y. Vardi. Undecidable boundedness problems for datalog programs. *J. Log. Program.*, 25(2):163–190, 1995. doi:10.1016/0743-1066(95)00051-K.

34 ISO/IEC 13211-1:1995. *Information technology – Programming languages – Prolog – Part 1: General core.* International Organization for Standardization, 1995.

35 Alex Ivliev, Lukas Gerlach, Simon Meusel, Jakob Steinberg, and Markus Krötzsch. Nemo: Your friendly and versatile rule reasoning toolkit. In Pierre Marquis, Magdalena Ortiz, and Maurice Pagnucco, editors, *Proc. 21st Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'24)*, pages 743–754. IJCAI Organization, 2024. `doi:10.24963/kr.2024/70`.

36 Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Proc. 28th Int. Conf. on Computer Aided Verification (CAV'16), Part II*, volume 9780 of *LNCS*, pages 422–430. Springer, 2016. `doi:10.1007/978-3-319-41540-6_23`.

37 Mark Kaminski, Bernardo Cuenca Grau, Egor V. Kostylev, Boris Motik, and Ian Horrocks. Foundations of declarative data analysis using limit datalog programs. In Carles Sierra, editor, *Proc. 26th Int. Joint Conf. on Artificial Intelligence (IJCAI'17)*, pages 1123–1130. ijcai.org, 2017.

38 Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. Datalog in wonderland. *SIGMOD Rec.*, 51(2):6–17, 2022. `doi:10.1145/3552490.3552492`.

39 Michael Kifer and Harold Boley, editors. *RIF Overview.* W3C Working Group Note, 22 June 2010. Available at `http://www.w3.org/TR/rif-overview/`.

40 Robert A. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Proc. 6th IFIP Congress*, pages 569–574. North-Holland, 1974. Preprint available online at `https://www.doc.ic.ac.uk/~rak/papers/IFIP%2074.pdf`.

41 Markus Krötzsch, Maximilian Marx, and Sebastian Rudolph. The power of the terminating chase. In Pablo Barceló and Marco Calautti, editors, *Proc. 22nd Int. Conf. on Database Theory (ICDT'19)*, volume 127 of *LIPIcs*, pages 3:1–3:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

42 Markus Langer, Daniel Oster, Timo Speith, Holger Hermanns, Lena Kästner, Eva Schmidt, Andreas Sesing, and Kevin Baum. What do we want from explainable artificial intelligence (XAI)? – A stakeholder perspective on XAI and a conceptual model guiding interdisciplinary XAI research. *Artif. Intell.*, 296:103473, 2021. `doi:10.1016/J.ARTINT.2021.103473`.

43 Nicola Leone, Carlo Allocca, Mario Alviano, Francesco Calimeri, Cristina Civili, Roberta Costabile, Alessio Fiorentino, Davide Fuscà, Stefano Germano, Giovanni Laboccetta, Bernardo Cuteri, Marco Manna, Simona Perri, Kristian Reale, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. Enhancing DLV for large-scale reasoning. In Marcello Balduccini, Yuliya Lierler, and Stefan Woltran, editors, *Proc. 15th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*, volume 11481 of *LNCS*, pages 312–325. Springer, 2019. `doi:10.1007/978-3-030-20528-7_23`.

44 Vladimir Lifschitz. *Answer Set Programming.* Springer, 2019.

45 Yanhong A. Liu and Scott D. Stoller. Recursive rules with aggregation: a simple unified semantics. *J. Log. Comput.*, 32(8):1659–1693, 2022. `doi:10.1093/LOGCOM/EXAC072`.

46 Logica Devs. *Logica project webpage*, Retrieved 04 Sept 2025. `https://logica-web.github.io/`.

47 Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From datalog to flix: a declarative language for fixed points on lattices. In Chandra Krintz and Emery D. Berger, editors, *Proc. 37th ACM SIGPLAN Conf. Programming Language Design and Implementatio (PLDI'16)*, pages 194–208. ACM, 2016. `doi:10.1145/2908080.2908096`.

48 David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. Datalog: concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, volume 20 of *ACM Books*, pages 3–100. ACM / Morgan & Claypool, 2018. `doi:10.1145/3191315.3191317`.

49 Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In Denny Vrandečić, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti,

Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *Proc. 17th Int. Semantic Web Conf. (ISWC'18)*, volume 11137 of *LNCS*, pages 376–394, 2018.

50 Bruno Marnette. Generalized schema-mappings: from termination to tractability. In Jan Paredaens and Jianwen Su, editors, *Proc. 28th Symposium on Principles of Database Systems (PODS'09)*, pages 13–22. ACM, 2009.

51 Maximilian Marx and Markus Krötzsch. Tuple-generating dependencies capture complex values. In Dan Olteanu and Nils Vortmeier, editors, *Proc. 25th Int. Conf. on Database Theory (ICDT'22)*, volume 220 of *LIPIcs*, pages 13:1–13:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ICDT.2022.13.

52 Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz, editors. *OWL 2 Web Ontology Language: Profiles.* W3C Recommendation, 2009. Available at http://www.w3.org/TR/owl2-profiles/.

53 Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. Incremental update of datalog materialisation: the backward/forward algorithm. In Blai Bonet and Sven Koenig, editors, *Proc. 29th AAAI Conf. on Artif. Intell. (AAAI'15)*, pages 1560–1568. AAAI Press, 2015. doi:10.1609/AAAI.V29I1.9409.

54 Marie-Laure Mugnier and Michaël Thomazo. An introduction to ontology-based query answering with existential rules. In Manolis Koubarakis, Giorgos B. Stamou, Giorgos Stoilos, Ian Horrocks, Phokion G. Kolaitis, Georg Lausen, and Gerhard Weikum, editors, *Proc. 10th Int. Reasoning Web Summer School (RW'14)*, volume 8714 of *LNCS*, pages 245–278. Springer, 2014. doi:10.1007/978-3-319-10587-1_6.

55 Inderpal Singh Mumick and Oded Shmueli. How expressive is statified aggregation? *Ann. Math. Artif. Intell.*, 15(3-4):407–434, 1995. doi:10.1007/BF01536403.

56 Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A highly-scalable RDF store. In Marcelo Arenas et al., editor, *Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II*, volume 9367 of *LNCS*, pages 3–20. Springer, 2015. doi:10.1007/978-3-319-25010-6_1.

57 Peter F. Patel-Schneider and David L. Martin. EXISTStential aspects of SPARQL. In Takahiro Kawamura and Heiko Paulheim, editors, *Proc. Posters & Demonstrations Track of the 15th International Semantic Web Conference (ISWC 2016)*, volume 1690 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

58 Robert Piro, Yavor Nenov, Boris Motik, Ian Horrocks, Peter Hendler, Scott Kimberly, and Michael Rossman. Semantic technologies for data analysis in health care. In Paul T. Groth, Elena Simperl, Alasdair J. G. Gray, Marta Sabou, Markus Krötzsch, Freddy Lécué, Fabian Flöck, and Yolanda Gil, editors, *Proc. 15th Int. Semantic Web Conf. (ISWC'16, Part II)*, volume 9982 of *LNCS*, pages 400–417, 2016. doi:10.1007/978-3-319-46547-0_34.

59 Eric Prud'hommeaux and Andy Seaborne, editors. *SPARQL Query Language for RDF.* W3C Recommendation, 15 January 2008. Available at http://www.w3.org/TR/rdf-sparql-query/.

60 Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In Moshe Y. Vardi and Paris C. Kanellakis, editors, *Proc. 11th Symposium on Principles of Database Systems (PODS'92)*, pages 114–126. ACM Press, 1992. doi:10.1145/137097.137852.

61 Sebastian Rudolph and Michaël Thomazo. Expressivity of datalog variants - completing the picture. In Subbarao Kambhampati, editor, *Proc. 25th Int. Joint Conf. on Artificial Intelligence (IJCAI'16)*, pages 1230–1236. AAAI Press, 2016.

62 Jiwon Seo, Stephen Guo, and Monica S. Lam. SociaLite: An efficient graph query language based on datalog. *IEEE Trans. Knowl. Data Eng.*, 27(7):1824–1837, 2015. doi:10.1109/TKDE.2015.2405562.

63 Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in DeALS. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *Proc. 31st IEEE Int. Conf. Data Engineering (ICDE'15)*, pages 867–878. IEEE Computer Society, 2015. doi:10.1109/ICDE.2015.7113340.

64 Johannes Tantow, Lukas Gerlach, Stephan Mennicke, and Markus Krötzsch. Verifying datalog reasoning with Lean. In Yannick Forster and Chantal Keller, editors, *Proc. 16th Int. Conf. Interactive Theorem Proving (ITP'25)*, volume 352 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

65 Jacopo Urbani, Ceriel Jacobs, and Markus Krötzsch. Column-oriented Datalog materialization for large knowledge graphs. In Dale Schuurmans and Michael P. Wellman, editors, *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16)*, pages 258–264. AAAI Press, 2016. `doi:10.1609/aaai.v30i1.9993`.

66 Jacopo Urbani, Markus Krötzsch, and Thomas Eiter. Chasing streams with existential rules. In *Proc. 19th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'22)*, pages 415–419, 8 2022. `doi:10.24963/kr.2022/43`.

67 Allen Van Gelder. Foundations of aggregation in deductive databases. In Stefano Ceri, Katsumi Tanaka, and Shalom Tsur, editors, *Proc. 3rd Int. Conf. Deductive and Object-Oriented Databases (DOOD'93)*, volume 760 of *LNCS*, pages 13–34. Springer, 1993. `doi:10.1007/3-540-57530-8_2`.

68 William Van Woensel, Dörthe Arndt, Pierre-Antoine Champin, Dominik Tomaszuk, and Gregg Kellogg, editors. *Notation3 Language*. W3C Draft Community Group Report, 15 May 2024. Available at `https://w3c.github.io/N3/spec/`.

69 Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10), 2014.

70 Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, Mark Kaminski, and Egor V. Kostylev. Datalogmtl: Computational complexity and expressive power. In Sarit Kraus, editor, *Proc. 28th Int. Joint Conf. on Artificial Intelligence (IJCAI'19)*, pages 1886–1892. ijcai.org, 2019. `doi:10.24963/IJCAI.2019/261`.

71 Dingmin Wang, Bernardo Cuenca Grau, Przemyslaw Andrzej Walega, and Pan Hu. Practical reasoning in DatalogMTL. *Theory Pract. Log. Program.*, 25(2):225–255, 2025. `doi:10.1017/S1471068424000164`.

72 Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory Pract. Log. Program.*, 17(5-6):1048–1065, 2017. `doi:10.1017/S1471068417000436`.

73 Eric Zhang. Percival project webpage. `https://percival.ink/`, Retrieved 04 Sept 2025.