Datalog Unchained

Victor Vianu
UC San Diego & Inria
vianu@ucsd.edu

ABSTRACT

This is the companion paper of a talk in the *Gems of PODS* series, that reviews the development, starting at PODS 1988, of a family of Datalog-like languages with procedural, forward chaining semantics, providing an alternative to the classical declarative, model-theoretic semantics. These languages also provide a unified formalism that can express important classes of queries including *fixpoint*, *while*, and all computable queries. They can also incorporate in a natural fashion updates and nondeterminism. Datalog variants with forward chaining semantics have been adopted in a variety of settings, including active databases, production systems, distributed data exchange, and data-driven reactive systems.

ACM Reference Format:
Victor Vianu. 2021. Datalog Unchained. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3452021.3458815

1 INTRODUCTION

Datalog emerged in the 80's as a simple, elegant formalism providing the basis for deductive databases and the study of recursion in database languages. Datalog was introduced as a specialization of logic programming to databases (a discussion of the history can be found in [95, 96]). Datalog$^\neg$, its extension with negation, was also studied through the lens of logic programming and *non-monotonic reasoning*, by which programs are viewed as specifications of consistent states of the world, best understood via model-theoretic semantics. This approach has led, most prominently, to *stratified* Datalog$^\neg$ [28, 50, 61, 92]. While stratified semantics provides a very natural interpretation for negation, it is limited to programs without recursion through negation. The quest to provide semantics to all Datalog$^\neg$ programs culminated with the elegant *well-founded* semantics [63]. However, its conceptual complexity, involving 3-valued models (or equivalently, alternating fixpoints), has precluded its widespread adoption in practice.

At PODS 1988, two articles [5, 87] independently suggested an alternative approach, aptly captured by the title of [87]: "Why not negation by fixpoint?". Both articles proposed to depart from the declarative approach and adopt a procedural semantics for Datalog$^\neg$ based on forward chaining of rules, called *inflationary fixpoint* semantics. As shown in [5], Datalog$^\neg$ with inflationary semantics combines simplicity with expressiveness: it captures precisely the robust class of *fixpoint* queries, previously defined by extensions of first-order logic with a fixpoint operator.

The two PODS 1988 articles initiated a line of research [3, 6, 8, 14, 15, 104, 116] that extended the forward chaining approach to an entire family of Datalog-like languages, providing an alternative paradigm for database queries, with natural counterparts to classical query languages including the *fixpoint* and *while* queries, and all the way to computable queries. The forward chaining approach turned out to also be suitable for studying nondeterministic languages, yielding an appealing unifying formalism. The present paper tells the story of this development.

After a brief review of classical query languages, we turn to Datalog and its extensions. We begin with an informal overview of Datalog and Datalog$^\neg$ with stratified and well-founded semantics. We then present the procedural, forward chaining semantics of Datalog$^\neg$. One of the nicest results in regard to expressive power is the convergence of the procedural and declarative semantics to the *fixpoint* queries. We also present a further extension denoted Datalog$^{\neg\neg}$ that allows for explicit retraction of facts and expresses the *while* queries. Finally, we discuss Datalog$^\neg_{new}$, a variant of Datalog that allows for the invention of new values in heads of rules, and expresses all computable queries. Value invention also arises in the object-oriented context, where object creation is a very useful and common feature [12].

Deterministic languages, including rule-based languages, have well-known limitations of expressive power. For example, there is no known language that expresses precisely the PTIME queries (e.g., see [79]). This limitation is overcome in the presence of an *order*. For example, Datalog$^\neg$ with inflationary or well-founded semantics expresses on ordered databases exactly the queries computable in polynomial time (and so do the weaker semi-positive and stratified Datalog$^\neg$). Another approach, intimately related to the first, trades off determinism for expressiveness. Indeed, as shown in [3, 8, 14], nondeterministic variants of Datalog with negation can express all (deterministic and nondeterministic) queries computable in polynomial time. The nondeterminism arises from the firing of rule instantiations in arbitrary order. As argued in [15], nondeterminism can be a very useful feature, independently of issues of expressiveness. Indeed, nondeterminism has long been present in expert systems and production systems (e.g., see [38]).

The paper is organized as follows. Some background is provided in Section 2. The declarative approach is briefly surveyed next. This reviews Datalog, stratified Datalog$^\neg$, and Datalog$^\neg$ with well-founded semantics. The procedural, forward chaining approach is presented in Section 4, together with results on the relative expressive power of the declarative and procedural languages. Nondeterministic languages are discussed in Section 5. A brief review of intervening Datalog research is outlined in Section 6. To conclude, we compare the forward chaining and declarative semantics for

PODS '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8381-3/21/06.
https://doi.org/10.1145/3452021.3458815

Datalog and discuss the adoption of the forward chaining semantics in practice. The presentation relies largely on material from [3, 5, 6, 8, 14, 15, 116]. The work surveyed in this paper is joint with Serge Abiteboul.

2 BACKGROUND

In this section, we review some terminology relating to relational databases. In particular, we recall some of the traditional query languages, including extensions of first-order logic with recursion (the *fixpoint* [49, 98] and *while* [47] queries).

We assume the existence of four infinite and pairwise disjoint sets of symbols: the set **rel** of *relation symbols*, the set **att** of *attributes*, the set **dom** of *constants*, and the set **var** of *variables*. A *relation schema* is a relation symbol together with a finite set of attributes. The set of attributes of a relation R is denoted by $att(R)$. A *database schema* is a finite set of relation schemas. A *free tuple* over a relational schema R is a mapping from $att(R)$ into $\mathbf{dom} \cup \mathbf{var}$. A *constant tuple* over a relational schema R is a mapping from $att(R)$ into **dom**. An *instance* over a relation schema R is a finite set of constant tuples over R. An *instance* $\mathbf{I}$ over a database schema $\mathbf{R}$ is a mapping from $\mathbf{R}$ such that for each R in $\mathbf{R}$, $\mathbf{I}(R)$ is an instance over R. The set of all instances over a schema $\mathbf{R}$ is denoted by $inst(\mathbf{R})$. Note that we only consider finite instances. For an instance $\mathbf{I}$, we denote by $adom(\mathbf{I})$ the set of elements of **dom** occurring in $\mathbf{I}$.

A *deterministic database query* is a mapping from $inst(\mathbf{R})$ to $inst(answer)$, where $\mathbf{R}$ is a database schema and *answer* is a relation schema not in $\mathbf{R}$. We also discuss nondeterministic queries. A *nondeterministic query* is a subset of $inst(\mathbf{R}) \times inst(answer)$ for some $\mathbf{R}$ and *answer* (so it is a relation rather than a mapping). Some settings use a more general notion of *database transformation* allowing answer schemas with several relations, some of which may be part of the input schema (thus capturing updates).

Queries are usually required to obey three conditions: *well-typedness, computability* and *genericity* [17, 48]. Well-typedness requires that the results of the query be instances of a *fixed* relation schema. A query is computable if there is a Turing machine that, given any standard encoding of an input database (depedent on a total order on the domain), produces a standard encoding of the query answer. Genericity requires that the graph of a database query be closed under isomorphisms of the domain that fix a specified finite set of constants depending on the query.

We will refer to complexity classes of queries. For each Turing machine complexity class C, there is a corresponding complexity class of (nondeterministic) queries denoted (N)DB-C. In particular, the class of nondeterministic queries which can be computed by a (nondeterministic) Turing machine in polynomial time is denoted (N)DB-PTIME. It is important to distinguish between classes NDB-C of nondeterministic queries and classes of deterministic queries defined using nondeterministic devices. For example, by Savitch's theorem, PSPACE = NPSPACE, so DB-PSPACE = DB-NPSPACE. Both are classes of deterministic queries. However, NDB-PSPACE contains nondeterministic queries, so DB-PSPACE ≠ NDB-PSPACE (and NDB-PSPACE ≠ DB-NPSPACE). Similarly, DB-NP is not to be confused with NDB-PTIME ! Below are some informal examples. All queries take as input a binary relation representing edges in a graph:

- The query whose answer consists of all vertices lying on a cycle is a query in DB-PTIME.
- The nondeterministic query whose answer is obtained by deleting one of the edges $\langle a, b\rangle$ or $\langle b, a\rangle$ for every cycle $\{\langle a, b\rangle, \langle b, a\rangle\}$ is a query in NDB-PTIME.
- The query whose answer is a unary relation which is empty if the graph has no Hamiltonian circuit and is the set of vertices of the graph otherwise, is in DB-NP (because testing for Hamiltonicity has complexity NP).

Some query languages. Most practical query languages in relational databases are based on FO, first-order logic on relations, sometimes called *relational calculus* (e.g. see [2]). FO has an algebraization called *relational algebra* [51]. Relational algebra provides the following operations on relations: π_X (projection on attributes X), σ_C (selection of tuples satisfying condition C consisting of (in)equalities among attributes and/or constants), $\delta_{A\to B}$ (rename attribute A to B), $\bowtie$ (join of two relations), $-$ (difference), and $\cup$ (union).

There are many useful queries that FO cannot express, such as the transitive closure of a graph. Numerous extensions of FO with recursion have been proposed. Most of them converge towards two very robust classes of queries: *fixpoint* [49, 98] and *while* [47]. These can be defined in various ways: by adding fixpoint operators to FO [6, 98], looping constructs to relational algebra [47, 49], or by extensions of Datalog [6]. We briefly review here the definition of *fixpoint* and *while* based on the eponymous languages, using looping constructs (see [2]).

While is an imperative language that extends FO with recursion. It provides relation variables, assignment statements of the form $R := \varphi$ where φ is an FO query, and a looping construct *while* φ *do* where φ is an FO sentence. An equivalent variation uses loops of the form *while change do* which iterate the body as long as some change is made to some relation. *Fixpoint* is the same as *while* except the semantics of assignment is cumulative (i.e., an assignment denoted $R \mathrel{+}= \varphi$ *adds* φ to the current content of R). This guarantees termination of *fixpoint* programs in polynomial time, whereas *while* programs require polynomial space. On ordered databases, *fixpoint* expresses *precisely* DB-PTIME [83, 114] and *while* expresses DB-PSPACE [114]. It was further shown in [7] that *fixpoint* = *while* iff PTIME = PSPACE, even without the order assumption.

3 THE DECLARATIVE APPROACH

We briefly review the classical model-theoretic semantics of Datalog and its extension with negation. For a detailed presentation see [2], and [96] for a more comprehensive survey.

3.1 Datalog

Much of the activity in deductive databases has focused on a toy language called *Datalog*. Some of the early history of Datalog is discussed in [95, 96]. Although limited, Datalog highlights some aspects of recursion present in many practical languages. Most of the optimization techniques in deductive databases have been developed around Datalog.

As an example, following is a Datalog program that computes the transitive closure of a graph. The graph is represented in relation

G and its transitive closure in relation T:

$$\begin{array}{lcl} T(x,y) & \leftarrow & G(x,y) \\ T(x,y) & \leftarrow & G(x,z), T(z,y). \end{array}$$

A Datalog program "defines" the relations occurring in heads of rules, from the other relations. The definition is recursive, so defined relations can also occur in bodies of rules. Thus, a Datalog program is interpreted as a mapping from instances over the relations occurring in the bodies only, to instances over the relations occurring in the heads. For example, the program above maps a relation over G (a graph) to a relation over T (its transitive closure).

We now define the syntax of Datalog.

Definition 3.1. A *(Datalog) rule* is an expression of the form:

$$R_1(u_1) \leftarrow R_2(u_2), \ldots, R_n(u_n)$$

where $n \geq 1$, $R_1, \ldots, R_n$ are relation names, and $u_1, \ldots, u_n$ are free tuples (tuples of variables and constants). Each variable occurring in u_1 must occur in at least one of $u_2, \ldots, u_n$. A *Datalog program* is a finite set of Datalog rules. The *head* of the rule is the expression $R_1(u_1)$; and $R_2(u_2), \ldots, R_n(u_n)$ forms the *body*.

The set of constants occurring in a Datalog program P is denoted $adom(P)$; and for an instance $\mathbf{I}$, we use $adom(P, \mathbf{I})$ as an abbreviation for $adom(P) \cup adom(\mathbf{I})$.

Let P be a Datalog program. An *extensional* relation is a relation occurring only in the body of the rules. An *intensional* relation is a relation occurring in the head of some rule of P. The *extensional (database) schema*, denoted $edb(P)$, consists of the set of all extensional relation names; whereas the *intensional schema* $idb(P)$ consists of all the intensional ones. The *schema* of P, denoted $sch(P)$ is the union of $edb(P)$ and $idb(P)$. The semantics of a Datalog program is a mapping from database instances over $edb(P)$ to database instances over $idb(P)$. Typically, one relation of $idb(P)$ is designated as the answer relation.

The key idea of the declarative approach of deductive databases is to view the program as a set of first-order sentences that describes the desired answer. To a Datalog rule

$$\rho : R_1(u_1) \leftarrow R_2(u_2), \ldots, R_n(u_n)$$

we can associate the logical sentence:

$$\forall x_1, \ldots, x_m (R_1(u_1) \leftarrow R_2(u_2) \wedge \ldots \wedge R_n(u_n))$$

where $x_1, \ldots, x_m$ are the variables occurring in the rule; and "$\leftarrow$" is the standard logical *implication*. For a program P, the set of sentences associated with the rules of P is denoted by Σ_P. It turns out that for each Datalog program P, and input $\mathbf{I}$, there is a minimum model of Σ_P extending $\mathbf{I}$. This model is the semantics of P on input $\mathbf{I}$ and is denoted by $P(\mathbf{I})$.

3.2 Stratified Datalog$^\neg$

Datalog$^\neg$ extends Datalog with negations in the bodies of rules. The syntax of Datalog$^\neg$ is a straightforward extensions of Datalog. A Datalog$^\neg$ *rule* is an expression of the form

$$R_1(u_1) \leftarrow L_1, ..., L_n$$

where: R_1 is a relation, u_1 a free tuple, and each L_i is a literal of the form $R_i(u_i)$ (in which case it is called *positive*) or $\neg R_i(u_i)$ (in which case it is called *negative*). Each variable in u_1 must occur in some literal L_i of the body.

A Datalog$^\neg$ *program* is a non-empty finite set of Datalog$^\neg$ rules. As for Datalog programs, $sch(P)$ denotes the database schema consisting of all relations involved in the program P; the relations occurring in heads of rules are the *idb* relations of P, and the others are the *edb* relations of P.

Similarly to Datalog, we can associate to a Datalog$^\neg$ program P the set Σ_P of FO sentences corresponding to the rules of P. For Datalog, the model-theoretic semantics of a program P is given by the unique minimal model of Σ_P extending the input. Unfortunately, this simple solution no longer works for Datalog$^\neg$, since uniqueness of a minimal model extending the input is not guaranteed. Short of this guarantee, a model-based semantics must specify an "intended" model that is intuitive and easy to compute. This is the core problem of "non-monotonic reasoning": make sense in a consistent way of the co-existence of negative and positive facts, in a way that reflects a natural reasoning process.

The most intuitive and widely accepted declarative semantics for Datalog$^\neg$ requires a syntactic restriction that, informally, prohibits recursion through negation. The resulting syntactic class is called *stratified* Datalog$^\neg$. Intuitively, the restriction allows to "read" the program so that, for each *idb* relation R, the portion of P defining R comes before the negation of R is used. Once R is computed, the set of negative facts over R (restricted to the active domain) is well defined. For example, consider the following stratified Datalog$^\neg$ program P defining the complement of transitive closure of a graph G:

$$\begin{array}{lcl} T(x,y) & \leftarrow & G(x,y) \\ T(x,y) & \leftarrow & G(x,z), T(z,y) \\ CT(x,y) & \leftarrow & \neg T(x,y). \end{array}$$

According to stratified semantics, the idb relation T is defined by the first two rules, and then its negation is used in the rule defining CT. Thus, the first two rules are applied before the third. This approach can be naturally extended to any stratified Datalog$^\neg$ program. Not surprisingly, this appealing semantics has been independently proposed by quite a few investigators [28, 50, 61, 92].

3.3 The Well-Founded Semantics

While stratification provides a simple and elegant approach to defining semantics of Datalog$^\neg$ programs, it has two major limitations. First, it does not provide semantics to *all* Datalog$^\neg$ programs. Second, stratified Datalog$^\neg$ programs are not quite satisfactory with regard to expressive power. From a computational point of view, they provide recursion and negation. Therefore, one might expect that they express the *fixpoint* queries. Unfortunately, as shown [86] (making use of earlier results from [53] and [49]), stratified Datalog$^\neg$ programs fall short of expressing all *fixpoint* queries. Intuitively, this is due to the fact that the stratification condition prohibits recursive application of negation, whereas in other languages expressing *fixpoint* this computational restriction does not exist.

The quest for declarative semantics for all Datalog$^\neg$ program has resulted in various proposals, of which the most prominent is the well-founded semantics [63]. It relies on a fundamental revision of the expectations on the answer to a Datalog$^\neg$ program. Previously,

the answer was required to provide information on the truth or falsehood of every fact. The well-founded semantics is based on the idea that a given program may not necessarily provide such information on all facts. Instead, some facts may simply be indifferent to it, and the answer should be allowed to say that the truth value of those facts is *unknown*. Relaxing expectations about the answer in this fashion allows to provide an elegant, model-theoretic semantics to *all* Datalog$^\neg$ programs. The price to pay is that the answer is no longer guaranteed to provide total information. In particular, the model-theoretic semantics requires a 3-valued logic.

Example 3.2. [63, 65] The example concerns a game with states, $a, b, \ldots$. The game is between two players. The possible moves of the games are held in a binary relation *moves*. A tuple $\langle a, b\rangle$ in *moves* indicates that when in state a, one can choose to move to state b. A player loses if they are in a state from which there are no moves. The goal is to compute the set of winning states, i.e., the set of states such that there exists a winning strategy for a player in this state. These are obtained in a unary predicate *win*.

Consider the input $\mathbf{K}$ with the following value for *moves*:

$$\mathbf{K}(moves) = \{\langle b, c\rangle, \langle c, a\rangle, \langle a, b\rangle, \langle a, d\rangle, \langle d, e\rangle, \langle d, f\rangle, \langle f, g\rangle\}$$

It is easily seen that there are indeed winning strategies from states d (move to e) and f (move to g). Slightly more subtle is the fact that there is no winning strategy from any of states a, b, or c. Indeed, a given player can prevent the other from winning, essentially by forcing a non-terminating sequence of moves.

Now consider the following nonstratifiable program P_{win}:

$$win(x) \quad \leftarrow \quad moves(x, y), \neg win(y)$$

Intuitively, P_{win} states that a state x is in *win* if there is at least one state y that one can move to from x, for which the opposing player looses. Following is a 3-valued model $\mathbf{J}$ of P_{win}, that agrees with $\mathbf{K}$ on *moves*, and is in fact the well-founded semantics of P_{win} on input $\mathbf{K}$. Instance $\mathbf{J}$ is such that $\mathbf{J}(moves) = \mathbf{K}(moves)$ and the values of *win*-atoms are given as follows:

true	$win(d), win(f)$
false	$win(e), win(g)$
unknown	$win(a), win(b), win(c)$.

The well-founded semantics can be compared against classical 2-valued model-theoretic semantics by casting its 3-valued semantics into a 2-valued model by taking the true facts as the answer to a program. In fact, it turns out that for every Datalog$^\neg$ program P there is a Datalog$^\neg$ program $\bar{P}$ whose well-founded semantics yields a classical 2-valued model whose true facts are the same as those of P under well-founded semantics [57, 58]. It was shown in [62] that with the 2-valued interpretation, well-founded semantics has the same expressive power as the *fixpoint* queries, and can also be evaluated in PTIME. This non-trivial result makes use of an equivalent formulation of the well-founded semantics as an *alternating fixpoint* computation [62]. Thus, well-founded semantics overcomes the expressiveness limitations of stratified Datalog$^\neg$.

Research on well-founded semantics, and the related notion of 3-stable model, has its roots in investigations of stable and default model semantics. Stable model semantics was introduced in [65] and default model semantics in [36, 37]. Stable semantics is based on Moore's autoepistemic logic [97], and default semantics is based on Reiter's default logic [108]. The equivalence between autoepistemic and default logic in the general case has been shown in [88]. The equivalence between stable model semantics and default model semantics was shown in [37].

Several equivalent definitions of the well-founded semantics have been proposed. The 3-valued model-theoretic approach is due to [106]. In addition to the alternating fixpoint computation of [62], other approaches for computing the well-founded semantics are exhibited in [37, 105]. Historically, the first definition of the well-founded semantics was proposed in [63, 64].

In summary, the stratified semantics provides a very natural, intuitive interpretation for the sublass of Datalog$^\neg$ without recursion through negation. While the well-founded semantics provides model-theoretic semantics to all Datalog$^\neg$ programs, it is far less intuitive and its widespread use remains unlikely. In terms of expressiveness, Datalog$^\neg$ with well-founded semantics overcomes the limitations of stratified Datalog$^\neg$ by capturing the *fixpoint* queries. At the time, no model-theoretic semantics had been defined for Datalog-like languages that yielded expressiveness beyond *fixpoint*. These limitations motivated the study of the forward chaining approach to Datalog-like languages, that had already been in practical use in production systems [38] and active databases [117].

4 THE FORWARD CHAINING APPROACH

We describe next the forward chaining semantics of Datalog$^\neg$ (expressing the *fixpoint* queries) and two extensions: Datalog$^{\neg\neg}$ (allowing negations in heads of rules and expressing the *while* queries) and Datalog$^\neg_{new}$ (allowing for the "invention" of new values, and expressing all computable queries).

4.1 Datalog$^\neg$

The forward chaining (or inflationary) semantics of Datalog$^\neg$ is intuitively very simple: the rules of the program are fired in parallel with all applicable instantiations, until a fixpoint is reached. We first illustrate this straightforward semantics with an example.

Example 4.1. We present a Datalog$^\neg$ program with input a graph in binary relation G. The program computes the relation $closer(x, y, x', y')$ defined as follows:

$$closer(x, y, x', y') = \{\langle x, y, x', y'\rangle \mid d(x, y) \leq d(x', y')\},$$

where $d(a, b)$ denotes the distance between nodes a and b in G. ($d(a, b)$ is infinite if there is no path from x to y.) The program is:

$$\begin{array}{lcl} T(x, y) & \leftarrow & G(x, y) \\ T(x, y) & \leftarrow & T(x, z), G(z, y) \\ closer(x, y, x', y') & \leftarrow & T(x, y), \neg T(x', y'). \end{array}$$

The program is evaluated as follows. The rules are fired simultaneously with all applicable valuations. At each such firing, some facts are inferred. This is repeated until no new facts can be inferred. A negative fact such as $\neg T(x', y')$ is true if $T(x', y')$ has not been inferred so far. This does not preclude $T(x', y')$ from being inferred at a later firing of the rules. One firing of the rules is called a "stage" in the evaluation of the program. In the above program, the transitive closure of G is computed in T. Consider the consecutive stages in the evaluation of the program. Note that, if the fact $T(x, y)$ is inferred at stage n, then $d(x, y) = n$. So, if $T(x', y')$ has not been inferred yet, this means that the distance between x and y is less than

that between x' and y'. Thus, if $T(x, y)$ and $\neg T(x', y')$ hold at some stage n, then $d(x, y) \leq n$ and $d(x', y') > n$ and $closer(x, y, x', y')$ is then inferred.

Formally, the forward chaining semantics of Datalog$^\neg$ is defined as follows. Let P be a Datalog$^\neg$ program and $\mathbf{K}$ an instance over $sch(P)$. An *instantiation* of a rule $A \leftarrow L_1, \ldots, L_n$ with respect to $\mathbf{K}$ is a rule $\nu(A) \leftarrow \nu(L_1), \ldots, \nu(L_n)$ where ν is a valuation which maps each variable into $adom(P, \mathbf{K})$. A fact A' is an *immediate consequence* for $\mathbf{K}$ and P if $A' \in \mathbf{K}(R)$ for some *edb* relation R, or $A' \leftarrow L'_1, ..., L'_n$ is an instantiation of a rule in P and: each positive L'_i is a fact in $\mathbf{K}$, and for each negative $L'_i = \neg A'_i$, $A'_i \notin \mathbf{K}$. The *immediate consequence operator* of P, denoted Γ_P, is now defined as follows. For each $\mathbf{K}$ over $sch(P)$,

$$\Gamma_P(\mathbf{K}) = \mathbf{K} \cup \{A \mid A \text{ is an immediate consequence for } \mathbf{K} \text{ and } P\}.$$

Given an instance $\mathbf{I}$ over $edb(P)$, one can compute $\Gamma_P(\mathbf{I}), \Gamma_P^2(\mathbf{I}), \Gamma_P^3(\mathbf{I})$, etc. As suggested in Example 4.1, each application of Γ_P is called a *stage* in the evaluation. From the definition of Γ_P, it follows that

$$\Gamma_P(\mathbf{I}) \subseteq \Gamma_P^2(\mathbf{I}) \subseteq \Gamma_P^3(\mathbf{I}) \subseteq \ldots$$

As for Datalog, the sequence reaches a fixpoint, denoted $\Gamma_P^\omega(\mathbf{I})$, after a finite number of steps. The restriction of this to the idb relations (or some subset thereof) is called the *image* (or *answer*) of P on $\mathbf{I}$.

In the procedural semantics described above, increasing sets of facts are inferred by firings of the rules. For that reason, this semantics is also referred to as the *inflationary* semantics for Datalog$^\neg$ (there is an "inflation" of tuples!). The language Datalog$^\neg$ with inflationary semantics is also referred to as *inflationary Datalog*$^\neg$. This semantics was first proposed for Datalog$^\neg$ in [5, 87]. It extends the fixpoint semantics proposed for Datalog in [50], also considered earlier in the context of logic programming [27, 113].

In the case of Datalog, the minimum model semantics and the inflationary fixpoint semantics coincide. For Datalog$^\neg$, this perfect match of declarative and procedural semantics is lost. First, a Datalog$^\neg$ program may not have a unique minimal model. Moreover, while the result produced by inflationary semantics is a model of the program, it is not necessarily a minimal one.

Datalog$^\neg$ with inflationary semantics provides recursion and negation, and it is straightforward to see that every query it expresses is a *fixpoint* query. The converse would seem unlikely, since Datalog$^\neg$ is a much simpler language than those previously known to express the *fixpoint* queries. Surprisingly, the following was shown in [5, 6]:

Theorem 4.2. *Inflationary Datalog$^\neg$ expresses precisely the* fixpoint *queries.*

The simulation of *fixpoint* by inflationary Datalog$^\neg$ presents two main difficulties, related to the simulation of the control capabilities available in *fixpoint*. The first involves delaying the firing of a rule until after the completion of a fixpoint by another set of rules. Intuitively, this is hard because checking that the fixpoint has been reached involves checking the *non-existence* rather than the existence of some valuation, and Datalog$^\neg$ is more naturally geared towards checking the *existence* of valuations. The solution to this difficulty is illustrated in the following example.

Example 4.3. The following Datalog$^\neg$ program computes the complement of the transitive closure of a graph G. The example illustrates the technique used to delay the firing of a rule (computing the complement) until the fixpoint of a set of rules (computing the transitive closure) has been reached, i.e., until the application of the transitivity rule yields no new tuples. To monitor this, the relations *old-T*, *old-T-except-final* are used. *old-T* follows the computation of T, but is one step behind it. The relation *old-T-except-final* is identical to *old-T*, but includes a clause which prevents it from firing when T has reached its last iteration. Thus, *old-T* and *old-T-except-final* differ only in the iteration after the transitive closure T reaches its final value. In the subsequent iteration, the program recognizes that the fixpoint has been reached, and fires the rule computing the complement in relation CT. The program is:

$$\begin{array}{lcl}
T(x, y) & \leftarrow & G(x, y) \\
T(x, y) & \leftarrow & G(x, z), T(z, y) \\
\textit{old-T}(x, y) & \leftarrow & T(x, y) \\
\textit{old-T-except-final}(x, y) & \leftarrow & T(x, y), T(x', z'), T(z', y'), \\
 & & \neg T(x', y') \\
CT(x, y) & \leftarrow & \neg T(x, y), \textit{old-T}(x', y'), \\
 & & \neg \textit{old-T-except-final}(x', y').
\end{array}$$

(It is assumed that G is not empty.)

The second difficulty concerns keeping track of iterations of the body in the computation of a loop. Given a loop

$$\textit{while change do body}$$

the simulation of *body* itself may involve numerous relations, whose behavior may be "sabotaged" by an overly zealous application of iteration. To overcome this we separate the "internal" computation of the body from the "external" iteration, as illustrated in the following example.

Example 4.4. Let G be a binary relation schema. Consider the *fixpoint* program

$$\begin{array}{l}
\textit{good} \mathrel{+}= \emptyset; \\
\textit{while change do} \\
\textit{good} \mathrel{+}= \varphi
\end{array}$$

where

$$\varphi = \forall y\, (G(y, x) \rightarrow \textit{good}(y)).$$

Note that the query computes the set of nodes in G that are not reachable from a cycle. (In other words, the nodes such that the lengths of paths leading to them are bounded.) One iteration is achieved by the Datalog$^\neg$ program P:

$$\begin{array}{lcl}
\textit{bad}(x) & \leftarrow & G(y, x), \neg \textit{good}(y) \\
\textit{delay} & \leftarrow & \\
\textit{good}(x) & \leftarrow & \textit{delay}, \neg \textit{bad}(x)
\end{array}$$

Simply iterating P does not yield the desired result. Intuitively, the relations *delay* and *bad*, which are used as "scratch paper" in the computation of a single iteration of the loop, cannot be re-initialized, and so cannot be effectively re-used to perform the computation of subsequent iterations.

To overcome this problem, we essentially create a version of P for each iteration The versions are distinguished by using "timestamps". The nodes themselves serve as timestamps. The timestamps marking iteration i are the values newly introduced in relation *good* at iteration $i - 1$. Relations *delay* and *delay-stamped* are used to delay the derivation of new tuples in *good* until *bad* and *bad-stamped*

(respectively) have been computed in the current iteration. The process continues until no new values are introduced in an iteration. The full program is the union of the three rules given above, which perform the first iteration, and the following rules, which perform the iteration with timestamp t:

$$\begin{array}{lcl} \textit{bad-stamped}(x,t) & \leftarrow & G(y,x), \neg \textit{good}(y), \textit{good}(t) \\ \textit{delay-stamped}(t) & \leftarrow & \textit{good}(t) \\ \textit{good}(x) & \leftarrow & \textit{delay-stamped}(t), \neg \textit{bad-stamped}(x,t). \end{array}$$

4.2 Datalog$^{\neg\neg}$

Recall that in Datalog$^{\neg}$ with inflationary semantics, a fact that has been inferred can never be retracted. Datalog$^{\neg\neg}$ allows explicit retraction of a previously inferred fact (thus, the semantics of Datalog$^{\neg\neg}$ is "noninflationary") [6][1]. Syntactically, this is done using negations in heads of rules, interpreted as deletions of facts. Moreover, input relations are allowed in heads of rules, thus providing the ability to perform updates. The resulting language is denoted by Datalog$^{\neg\neg}$, to indicate that negations are allowed in both heads and bodies of rules.

The immediate consequence operator Γ_P and semantics of a Datalog$^{\neg\neg}$ program are analogous to those for Datalog$^{\neg}$ with the following important proviso. If a negative literal $\neg A$ is inferred, the fact A is removed, unless A is also inferred in the same firing of the rules. This gives priority to inference of positive over negative facts and is somewhat arbitrary. Other possibilities are: (i) give priority to negative facts, (ii) interpret the simultaneous inference of A and $\neg A$ as a "no-op", i.e., including A in the new instance only if it is there in the old one; and (iii) interpret the simultaneous inference of A and $\neg A$ as a contradiction which makes the result undefined. The chosen semantics has the advantage over (iii) that the result is always defined. In any case, the choice of semantics is not crucial: it results in equivalent languages. With the semantics chosen above, termination is no longer guaranteed. For instance, the program

$$\begin{array}{lcl} T(0) & \leftarrow & T(1) \\ \neg T(1) & \leftarrow & T(1) \\ T(1) & \leftarrow & T(0) \\ \neg T(0) & \leftarrow & T(0) \end{array}$$

never terminates on input $T(0)$. Indeed, the value of T flip-flops between $\{\langle 0\rangle\}$ and $\{\langle 1\rangle\}$ so no fixpoint is reached.

By a method similar to the above, it can be shown that Datalog$^{\neg\neg}$ expresses precisely the *while* queries [6].

How about the relationship between Datalog$^{\neg}$ and Datalog$^{\neg\neg}$? Clearly, Datalog$^{\neg}$ is subsumed by Datalog$^{\neg\neg}$. Intuitively, it is tempting to believe that Datalog$^{\neg\neg}$ is more powerful than Datalog$^{\neg}$ (considering only terminating queries). However, this is far from obvious. Indeed, the following is a consequence of results in [7] on the relationship between *fixpoint* and *while*:

Theorem 4.5. *Inflationary Datalog$^{\neg}$ is strictly less expressive than Datalog$^{\neg\neg}$ iff* PTIME ≠ PSPACE.

Since it is open whether PTIME ≠ PSPACE, the relationship between inflationary (or well-founded) Datalog$^{\neg}$ and Datalog$^{\neg\neg}$ remains open. However, the strict inclusion is conjectured to be true.

[1]The language Datalog$^{\neg\neg}$ is denoted in [6] by Datalog$^{\neg\star}$

4.3 Datalog$^{\neg}_{new}$

All the languages considered so far express queries within DB-PSPACE. Intuitively, this is so because each program uses a fixed number of relations of fixed arity, which are filled in the course of the computation with tuples over the elements of the input. Thus, such programs can build an amount of space which is no more than polynomial in the number of elements of the input.

Suppose that we wish to have a *complete* language, i.e. a language expressing all queries. One way to break the polynomial "space barrier", first suggested in [4], is to allow programs to invent new values in the course of the computation. Besides the computational justification, this is useful in object-oriented databases, where the creation of new object identifiers is a useful and very common feature. Such a feature was studied in the object-oriented language IQL [12].

We describe an extension of Datalog$^{\neg}$, defined in [6], that expresses all deterministic queries. This extension introduces new values in the course of the computation (but not in the answer). We informally describe the language, denoted by Datalog$^{\neg}_{new}$. The syntax is the same as that of Datalog$^{\neg}$, except that variables that do not appear in body of a rule may appear in its head. The inflationary semantics of this language is similar to that of Datalog$^{\neg}$. The only difference consists in the use of the variables that occur only in heads of rules: these are valuated *outside* the current active domain, thus resulting in the "invention" of new values. Specifically, in the application of the immediate consequence operator, each instantiation of a rule body in the current active domain is extended with *one* instantiation of the remaining variables with distinct values outside the active domain. The choice of the particular new values is nondeterministic. However, this is the *only* source of nondeterminism. If the final result contains only values from the input, which can be enforced by a straightforward syntactic safety restriction, then the query defined by such a program is deterministic. Furthermore, Datalog$^{\neg}_{new}$ is complete:

Theorem 4.6. *[6] Datalog$^{\neg}_{new}$ expresses all deterministic queries.*

Intuitively, the proof uses the invented values to simulate the Turing machine computing the query, as well as the encodings of the input database on the initial Turing tape, and the decoding of the output. Each total order of the active domain generates one standard encoding, and the computation is carried out in parallel on all the encodings. The new values provide the unbounded amount of space needed for the simulation, thus overcoming the PSPACE barrier.

The relative expressive power of the various rule-based languages is summarized in Figure 1. The arrow ⇑ indicates strict inclusion and the arrow ↑ indicates strict inclusion iff PTIME ≠ PSPACE.

We note that Datalog extensions with forward chaining semantics that model various active databases are investigated in [104]. The results provide insight into the programming paradigm of active databases, the interplay of various features, and their impact on expressiveness and complexity. In particular, this yields highly expressive active database languages capturing PSPACE, EXPTIME, EXPSPACE, as well as all computable queries, on ordered databases.

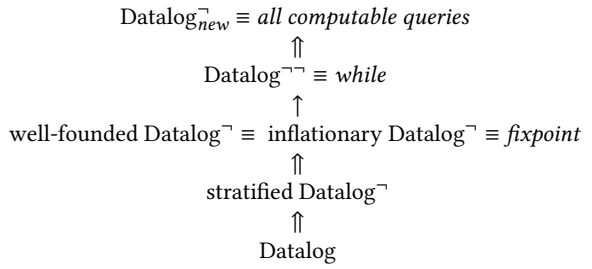

Figure 1: Relative expressive power of Datalog variants.

4.4 Limitations in expressive power

Although the languages discussed here are quite powerful, they have certain shortcomings with regard to expressive power. Indeed, there are very "simple" queries that none of the languages can express. The prototypical example is the evenness query on a unary relation:

$$even(R) = \begin{cases} true & \text{if } |R| \text{ is even} \\ false & \text{if } |R| \text{ is odd.} \end{cases}$$

where $|R|$ is the number of elements in R.

This difficulty is not specific to rule-based languages. Indeed, it extends to most deterministic languages. To understand the difficulty involved, consider the natural way to compute the query: remove elements from R one at a time, and keep a binary counter. However, the elements of R are logically undistinguishable one from another, so no deterministic language that adheres to the data independence principle can perform the algorithm just described. There are two ways out: (i) sacrifice data independence, or (ii) sacrifice determinism by allowing a nondeterministic construct to pick an arbitrary element from a set. We will explore each of these tradeoffs in turn. Suspending the data independence principle is modeled by access to an *order* among the elements in the database (which is a reasonable mathematical metaphor for access to the additional symmetry-breaking information provided by the internal storage).

Intuitively, there is a strong connection between the use of order (i.e., information on the internal storage) and nondeterminism. If a query is implemented using information on internal storage, then the answer may depend on such information and thus appear nondeterministic at the logical level.

4.5 The impact of order

The assumption that databases are ordered can have dramatic impact on the expressive power of languages. In ordered databases, the schema is assumed to contain a binary relation providing a total order on the active domain of each instance. With this assumption, it turns out that stratified Datalog¬, inflationary Datalog¬, and Datalog¬ with well-founded semantics are all equivalent and express precisely DB-PTIME. Furthermore, the apparently much weaker semi-positive Datalog¬, consisting of Datalog¬ where negation is only applied to edb relations, is almost as powerful as these languages. The "almost" is due to a technicality concerning the order: we also need to assume that the minimum and maximum constants are explicitly given. Surprisingly, these constants, that can be computed with an FO query if an order is given, cannot be computed with semi-positive programs.

THEOREM 4.7. *Stratified Datalog¬, Datalog¬ with well-founded semantics, and inflationary Datalog¬ are equivalent on ordered databases and express exactly the* DB-PTIME *queries. They are also equivalent to semi-positive Datalog¬ on ordered databases with min and max and express exactly the* DB-PTIME *queries.*

The result that semi-positive Datalog¬ expresses DB-PTIME on ordered databases with *min* and *max* is due to [101]. The result that inflationary Datalog¬ expresses DB-PTIME follows from results in [6] (equivalence of inflationary Datalog¬ and *fixpoint*) and [83, 114] (who showed that *fixpoint* expresses DB-PTIME on ordered databases).

Without the order assumption, there is no known deterministic language that expresses precisely the DB-PTIME queries. Indeed, the existence of such a language remains one of the main open problems in the theory of query languages (e.g., see [79]).

The following characterizes the power of Datalog¬¬ on ordered databases. It follows from a result of [6] showing the equivalence of Datalog¬¬ and *while*, and from a result of [114] showing that *while* expresses DB-PSPACE on ordered databases.

THEOREM 4.8. *Datalog¬¬ expresses exactly the* DB-PSPACE *queries on ordered databases.*

5 NONDETERMINISTIC LANGUAGES

The arguments in favor of nondeterministic languages are both practical and theoretical. The first is that nondeterminism occurs naturally in many practical settings. There are natural nondeterministic queries and updates, whose implementation using deterministic languages is contrived and inefficient. There are well-known applications in Artificial Intelligence which naturally lead to nondeterminism, and expert systems shells (such as KEE or OPS5 [39, 59]) whose rule-based components work nondeterministically. Reactive systems, such as data-driven workflows, are usually nondeterministic (e.g., see [10, 11, 16, 78]). The theoretical arguments for nondeterminism involve primarily the expressive power of nondeterministic languages. Indeed, the use of nondeterminism circumvents some of the expressiveness limitations discussed above, associated with deterministic languages. As mentioned, it is conjectured that there is no deterministic language expressing exactly the queries computable in polynomial time. On the other hand, there are nondeterministic languages expressing exactly the (deterministic and nondeterministic) queries computable in polynomial time.

5.1 Nondeterministic Datalog$^{\neg(\neg)}$

We next consider nondeterministic versions of the Datalog$^{\neg(\neg)}$ languages. Recall that the procedural, deterministic semantics for these languages was the result of evaluating programs by repeatedly firing all rules *in parallel*, up to a fixpoint. The nondeterministic semantics is obtained by firing one instantiation of a rule at a time, based on a nondeterministic choice. For instance consider the program:

$$\neg G(x, y) \leftarrow G(x, y), G(y, x).$$

With deterministic semantics, the program removes from the graph G all cycles of length two. With the nondeterministic semantics, the program computes one of several possible "orientations" for G

(i.e., for every pair of edges (x, y) and (y, x) in G, one of the edges is removed).

We first define the syntax of the nondeterministic version of Datalog$^{\neg\neg}$, denoted N-Datalog$^{\neg\neg}$ [6]. The difference with the deterministic version is that heads of rules may contain several literals, and equality can be used in bodies. It can be seen that these features would be redundant with the deterministic semantics.

Definition 5.1. A N-Datalog$^{\neg\neg}$ program is a finite set of rules of the form

$$A_1,...,A_k \leftarrow B_1,...,B_n$$

($k \geq 1$, $n \geq 0$), where each A_j is a literal of the form ($\neg$) $Q(x_1,...,x_m)$ ($m \geq 0$), and each B_i is a literal of the same form, or ($\neg$) $x_1 = x_2$ (the x_i's are variables or constants). It is required that each variable occurring in the head of a rule also occur positively bound in the body. □

To formally define the nondeterministic semantics, we introduce the notion of (nondeterministic) immediate successor of a set of facts using a rule. Let r be an N-Datalog$^{\neg\neg}$ rule. Let I be a set of facts and r' be an instantiation of r such that (i) each literal of the body of r' is true in I, (ii) the head of r' is consistent and (iii) each variable is valuated to some constant occurring in I. Then the instance J obtained from I by deleting the facts A such that $\neg A$ is in the head of r', and inserting the facts A in the head of r', is called an *immediate successor* of I using r.

By condition (ii) above, an instantiation of a rule is not considered if its head contains a literal and its negation.

Definition 5.2. Let P be an N-Datalog$^{\neg\neg}$ program. The *effect* of P is a relation over sets of facts defined as follows: for each I, (I, J) is in *eff(P)* iff there exists a sequence $I_0 = I, ..., I_n = J$ such that (i) for each i, I_{i+1} is an immediate successor of I_i using some r in P, and (ii) there is no immediate successor $J' \neq J$ of J using some rule in P. □

The language N-Datalog$^{\neg}$ is a specialization of N-Datalog$^{\neg\neg}$ obtained by disallowing negative literals in heads of rules (thus, negation can only occur in bodies of rules).

5.2 Expressive power

We present several results on the expressive power of N-Datalog$^{\neg}$ and N-Datalog$^{\neg\neg}$. As discussed earlier, the most significant result involves expressibility of NDB-PTIME.

We consider first the expressive power of N-Datalog$^{\neg\neg}$.

THEOREM 5.3. *[6] N-Datalog$^{\neg\neg}$ expresses exactly* NDB-PSPACE.

Consider next the expressive power of N-Datalog$^{\neg}$. It is easy to see that each N-Datalog$^{\neg}$ query is in NDB-PTIME. It turns out that there are simple NDB-PTIME queries that cannot be expressed in N-Datalog$^{\neg}$. We show this next, and then show how N-Datalog$^{\neg}$ can be augmented to increase the expressive power to NDB-PTIME.

The strict inclusion in NDB-PTIME is shown using the following example [6].

Example 5.4. Let $\mathbf{R}$ = {$P(A)$,$Q(AB)$}. It can be shown that there is no N-Datalog$^{\neg}$ program which computes $P - \pi_A(Q)$.

The precise characterization of the power of N-Datalog$^{\neg}$ is open. We note, however, that N-Datalog$^{\neg}$ expresses exactly NDB-PTIME in the presence of order.

As seen in the example above, there are very simple queries that N-Datalog$^{\neg}$ cannot compute. We now look at the origin of this weakness and show how it can be corrected. Note that N-Datalog$^{\neg}$ does not provide sufficient control capability to simulate the composition of two programs. Indeed, $P - \pi_A(Q)$ can be obtained as the composition of the mappings defined by the following two rules:

$$\begin{array}{lcl} T(x) & \leftarrow & Q(x, y), \text{ and} \\ answer(x) & \leftarrow & P(x), \neg T(x). \end{array}$$

The weak control capability of N-Datalog$^{\neg}$ makes it impossible for programs in this language to simulate the explicit control necessary to compute NDB-PTIME queries. Note that, in the case of N-Datalog$^{\neg\neg}$, the control needed is provided by deletions. For example, the query in Example 5.4 is computed by the following N-Datalog$^{\neg\neg}$ program:

$$\begin{array}{lcl} answer(x) & \leftarrow & P(x) \\ \neg answer(x), \neg P(x) & \leftarrow & Q(x, y). \end{array}$$

The constructs we add to N-Datalog$^{\neg}$ essentially provide sufficient control to simulate composition (in an inflationary manner). We consider two alternative constructs. The first construct allows for an "inconsistency" symbol $\perp$ to appear in heads of rules. The resulting language is denoted N-Datalog$^{\neg}\perp$. The idea is that if such a symbol is derived in a computation, that particular computation is abandoned. The second construct is universal quantification in bodies of rules and yields the language N-Datalog$^{\neg}\forall$. We first present N-Datalog$^{\neg}\perp$, then N-Datalog$^{\neg}\forall$. These languages are from [6].

N-Datalog$^{\neg}\perp$: The language N-Datalog$^{\neg}$ is extended with the symbol $\perp$ that can occur only as a literal in the head of rules. A pair (I,J) is in the effect of a N-Datalog$^{\neg}\perp$ program iff J is obtained by a computation where $\perp$ is not derived.

N-Datalog$^{\neg}\forall$: The language N-Datalog$^{\neg}$ is extended to allow rules of the form:

$$A_1, ..., A_q \leftarrow \forall \vec{x} B_1, ..., B_n,$$

where $\vec{x}$ is a sequence of variables occurring *only* in the body of the rule. Let $\vec{y}$ be the vector of the variables occurring in $B_1, ..., B_n$ and not in $\vec{x}$, and v be a valuation of $\vec{y}$. The rule is fired with valuation v if for each extension $\overline{v}$ of v to the variables in $\vec{x}$ (which valuates variables in $\vec{x}$ in the active domain), $\overline{v}B_1 \wedge ... \wedge \overline{v}B_n$ holds.

To illustrate these two languages, we show how to compute the query of Example 5.4 with N-Datalog$^{\neg}\forall$ or N-Datalog$^{\neg}\perp$ programs.

Example 5.5. The mapping $P - \pi_A$(Q) is computed by the following N-Datalog$^{\neg}\forall$ program:

$$answer(x) \leftarrow \forall y P(x), \neg Q(x, y).$$

A N-Datalog$^{\neg}\perp$ program computing the same query is:

$$\begin{array}{lcl} PROJ(x) & \leftarrow & \neg done\text{-}with\text{-}proj, Q(x, y) \\ done\text{-}with\text{-}proj & \leftarrow & \\ \perp & \leftarrow & done\text{-}with\text{-}proj, Q(x, y), \neg PROJ(x) \\ answer(x) & \leftarrow & done\text{-}with\text{-}proj, P(x), \neg PROJ(x). \end{array}$$

Intuitively, in N-Datalog$^\neg\forall$, one can check that a stage is completed (using $\forall$) before proceeding to the next one; this allows simulating composition. In N-Datalog$^\neg\bot$, a detected error leads to the derivation of $\bot$. The following shows that in fact these constructs provide sufficient power to bridge the gap between N-Datalog$^\neg$ and NDB-PTIME.

THEOREM 5.6. *[6] For each query τ the following are equivalent:*

- *τ is in* NDB-PTIME,
- *τ is defined by a N-Datalog$^\neg\bot$ program, and*
- *τ is defined by a N-Datalog$^\neg\forall$ program.*

We have seen so far nondeterministic languages capturing NDB-PSPACE and NDB-PTIME. Similarly to the deterministic case, a complete language can be obtained by augmenting N-Datalog$^\neg$ with the ability to create new values. The resulting language is denoted N-Datalog$^\neg_{new}$.

THEOREM 5.7. *[6] N-Datalog$^\neg_{new}$ expresses all nondeterministic queries.*

It turns out that the nondeterministic languages described above are closely related to nondeterministic extensions of fixpoint logics, introduced in [14] (see also [15]). In these languages, nondeterminism is provided by a *witness* operator W. Informally, $W\bar{x}\varphi(\bar{x})$ results in nondeterministically choosing a value of $\bar{x}$ that satisfies $\varphi(\bar{x})$. The addition of W to fixpoint logics yields nondeterministic extensions FO + IFP + W of inflationary fixpoint logic, and FO + PFP + W of partial fixpoint logic. It is shown in [14] that N-Datalog$^{\neg\neg}$ is equivalent to FO + PFP + W, and N-Datalog$^\neg\forall$ (as well as N-Datalog$^\neg\bot$) is equivalent to FO + IFP + W.

We note that another way to introduce nondeterminism in rule-based languages is provided by the *choice* operator first presented in [90]. This construct has been included in the language LDL, an implementation of Datalog$^\neg$ [99]. Variations of the choice operator, and its connection with stable models of Datalog$^\neg$ programs, are further studied in [66, 109]. The expressive power of the choice operator in the context of Datalog is investigated in [52]. The main result exhibits a language expressing exactly NDB-PTIME. Further Datalog languages using the choice operator and expressing the Boolean hierarchy are exhibited in [76].

5.3 Connections with Determinism

Recall that one of the motivations for considering nondeterministic languages is their ability to express more *deterministic* queries. In this section we consider the ability of various nondeterministic languages to express deterministic queries. We also consider the deterministic queries definable by considering the "possible" and "certain" answers of a nondeterministic query, in the spirit of queries on databases with incomplete information. The results are from [3].

The notion of *deterministic fragment* expressed by a language is defined next.

Definition 5.8. The *deterministic fragment* of a (nondeterministic) language is the set of deterministic queries defined by programs in the language. The deterministic fragment of a language L is denoted $det(L)$. □

In the previous section we characterized the nondeterministic queries expressible in the various languages. These results can be used to characterize the deterministic fragments expressible in these languages. Thus, we have:

THEOREM 5.9.
(1) det(N-Datalog$^\neg\forall$) = det(N-Datalog$^\neg\bot$) = DB-PTIME.
(2) det(N-Datalog$^{\neg\neg}$) = DB-PSPACE.

It is important to note that (1) does *not* provide a language expressing DB-PTIME, since it is undecidable whether a N-Datalog$^\neg\forall$ or N-Datalog$^\neg\bot$ program defines a deterministic query.

An alternative way of obtaining deterministic queries using nondeterministic programs is suggested by the work of [82] on incomplete information. Indeed, there is a natural connection between incomplete information and nondeterminism. As noted in [1], incomplete information can be seen as resulting from incompletely specified (therefore nondeterministic) updates. The notions of *possible* and *certain* answers in [82] suggest the following definition:

Definition 5.10. Given a nondeterministic program P, the image of an input $\mathbf{I}$ under P with the *possibility* semantics (denoted $poss(\mathbf{I}, P)$) and the *certainty* semantics (denoted $cert(\mathbf{I}, P)$) are defined by:

$$poss(\mathbf{I}, P) = \bigcup\{\mathbf{J} \mid (\mathbf{I}, \mathbf{J}) \in \mathit{eff}(P)\}, \text{ and}$$
$$cert(\mathbf{I}, P) = \bigcap\{\mathbf{J} \mid (\mathbf{I}, \mathbf{J}) \in \mathit{eff}(P)\}.$$

The deterministic queries expressed by a program P under possibility semantics is denoted *poss(P)*, and under certainty semantics *cert(P)*. For a language L,

$$poss(L) = \{poss(P) \mid P \in L\} \text{ and } cert(L) = \{cert(P) \mid P \in L\}.$$

The *poss* or *cert* semantics yield significant power:

THEOREM 5.11.
(1) poss(N-Datalog$^\neg\forall$) = poss(N-Datalog$^\neg\bot$) = DB-NP.
(2) cert(N-Datalog$^\neg\forall$) = cert*(N-Datalog$^\neg\bot$) =* DB-CO-NP.
(3) cert*(N-Datalog$^{\neg\neg}$) =* poss*(N-Datalog$^{\neg\neg}$) =* DB-PSPACE.

Observe that, for N-Datalog$^{\neg\neg}$, the *poss* and *cert* semantics do not yield additional power. In particular, these semantics can be simulated within the deterministic fragment of N-Datalog$^{\neg\neg}$.

This concludes our review of the Datalog-like languages with forward chaining semantics, studied in [3, 5, 6, 8, 14, 15, 116]. We next discuss briefly some of the developments in Datalog research since these languages have been proposed, and situate them in the broader context.

6 DATALOG REDUX

Over time, research on recursive queries has been received by the database systems community with varying degrees of enthusiasm. By the end of the 80's, the attitude towards Datalog research had turned largely negative. This is perhaps best exemplified by the influential Laguna Beach report of 1989 [102], which includes this statement: *"The participants were unanimously negative on the prospective research contribution of general recursive query processing, and interfaces between a DBMS and Prolog."* As late as 1998, the no less influential [111], stated that *"No practical applications of recursive query processing have been found to date"*.

And yet they persisted. Despite the ups-and-downs in popularity, work on Datalog-like languages continued in the database and logic programming communities in the 80's and 90's. On the theoretical front, there was much work on the expressiveness and complexity of Datalog-like languages (for a comprehensive survey, see [54]). Driven by theoretical and practical considerations, many extensions of Datalog have been put forward. They include arithmetic, sets, disjunction, aggregation, constraints, object-oriented constructs, complex objects, updates, etc. This has resulted in an entire ecosystem of languages, many of which were implemented. An excellent survey of these developments is provided in [96].

Research on Datalog was eventually rehabilitated in the database systems community and beyond. In an invited PODS 2010 talk [80] and companion paper [81], Joe Hellerstein delivered an impassioned rehabilitation of Datalog from the viewpoint of the systems community, pointing out the widespread use and effectiveness of Datalog-like languages in areas as diverse as security and privacy protocols, program analysis, natural language processing, probabilistic inference, modular robotics, multiplayer games, telecom diagnosis, networking, and distributed systems.

The last two decades have seen a robust "Datalog Spring" in which Datalog research has expanded with renewed energy. The sequence of Datalog workshops on the theme of the "Resurgence of Datalog in Academia and Industry" [20, 31, 55] was emblematic of this optimism. A broad account of this renewed activity, within and beyond the database community, can be found in [96]. Notable in the database theory community has been an increased symbiosis between theoretical research and practice (even yielding a number of start-ups!), as well as increased presence in adjacent areas such as AI and Knowledge Representation. Following are a few representative examples.

Datalog for networking Networking is an area where recursive processing and reasoning occurs naturally, and has led to the rise of "declarative networking", in which Datalog variants are used to specify and reason about distributed protocols and services (e.g., see [81, 93, 94]). These languages extend Datalog with a variety of features, including location identifiers, timestamps, and states (in the spirit of Statelog [91]). Dedalus and Bloom, both Datalog variants, were put forward by Hellerstein's team as a promising foundation for programming and reasoning about distributed systems [18, 19]. Their investigation of consistency properties of distributed systems led to the so called "CALM conjecture", [80, 81] which was subsequently studied in the database theory community using a model of communicating transducers [21–25]. The use of the declarative approach in network monitoring was also explored in [9]. It should be noted that the term "declarative" refers broadly to Datalog-like rule-based languages, with a mix of declarative and procedural semantics (e.g., see [103]). Negation, when present, is typically restricted to be stratified (and interpreted under stratified semantics). See [94] for a survey on declarative networking circa 2012.

Datalog for data extraction The Lixto project [32, 69] is a poster child for successful practical applications of Datalog relying on significant theoretical foundations. Lixto focuses on Web data extraction and has at its core Monadic Datalog over trees. The approach relies on deep and elegant theoretical results on the expressiveness and complexity of Monadic Datalog on trees and structures of bounded tree width [67, 68, 70–74]. In particular, it is shown in [68] that Monadic Datalog captures exactly Monadic Second Order logic over trees. This provides the expressiveness needed by wrappers for Web data extraction, while also guaranteeing efficiency. The Lixto project has been incorporated in a commercial product, McKinsey Periscope. A follow-up project, DIADEM, aimed to achieve fully automatic wrapper generation and uses at its core a related language based on Datalog [60].

Datalog for ontologies Datalog has proven to be an effective formalism for specifying and answering queries over ontologies. In particular, it provides an elegant unifying formalism that subsumes well-known description logics. Datalog+/- is a family of Datalog variants put forward for this purpose [40–42]. The Datalog+/- languages are obtained by first extending Datalog with existentially quantified variables in heads of rules, then considering various restrictions (guarded, linear, and weakly guarded) to ensure tractability. Other extensions, such as falsity in heads of rules, keys, and stratified negation, are also considered, and their complexity characterized. Importantly, the Datalog+/- family of languages can express the popular DLite family [44, 45] of descriprion logics. It also subsumes F-logic Lite [43], a tractable fragment of the well-known F-logic [85], central in deductive object-oriented databases.

Datalog for knowledge graphs An outgrowth of Datalog+/-, Vadalog is a Datalog-based language geared towards reasoning about knowledge graphs [33–35]. Vadalog has at its core Warded Datalog+/-, a variant of Datalog+/- that has good properties in terms of expressiveness and complexity [30, 75]. Vadalog is part of the VADA project [89, 112], in which Vadalog is extended with various features, including access to external resources such as text processing, data analytics and machine learning, modeled uniformly as transducers. The VADA project has generated considerable interest from industrial partners. Data analytics on big graphs is also provided by the BigDatalog system [110] that relies on Datalog extended with aggregates, with fixpoint semantics [118].

Datalog for data exchange on the Web Datalog variants are used in several approaches for high-level specifications of data sharing and exchange on the Web. Orchestra [78, 84] is a collaborative data sharing system that supports the exchange of data and updates among peers and relies on an extension of Datalog with Skolem functions. Webdamlog is a Datalog variant used to exchange data among peers on the Web, with the novel twist that *rules* can also be exchanged, in addition to data [11]. Webdamlog also uses updates in heads of rules, similarly to N-Datalog$^{\neg\neg}$ (see Section 4.2). The semantics is nondeterministic and based on forward chaining, similarly to active rules. Interestingly, the abilty to exchange rules is more than syntactic sugar: it increases the expressiveness of the language. The exchange of rules is also used in Active XML, whose core is a rule-based language using tree patterns [10, 13].

Datalog for data management Several startups have emerged that provide full data management systems relying on Datalog

variants. LogicBlox is a commercial database system that supports sophisticated analytics and is implemented on top of LogiQL, a rich extension of Datalog [29]. LogiQL can support business applications, workflows and data analytics, and is evaluated bottom-up. LogicBlox also contains updates needed for interactive features, in the spirit of Datalog$^{\neg\neg}$ [77]. Interestingly, the LogicBlox team included a substantial number of database theoreticians and produced some remarkable algorithms, such as the leapfrog trie join, shown to be optimal [115] in the sense of [100]. DATOMIC is another startup that has built a full data management system driven by a Datalog-like language [26].

7 CONCLUSION

We have reviewed the procedural, forward chaining approach to Datalog developed in [3, 5, 6, 8, 14, 15, 116], and briefly the state of contemporaneous Datalog research and later developments.

How does the procedural, forward chaining semantics measure up against the declarative alternative? The two paradigms are philosophically very different. The declarative approach attempts to model a natural reasoning process. In particular, consistency in the reasoning process is required: one cannot use a fact and later infer its negation, as can happen in the procedural semantics. As we have seen, the solution is ideally described as a model of the program satisfying certain desirable properties, such as minimality. However, in the presence of negation, the lack of a unique minimal model leads to the need to specify an "intended" model among several possible candidates. Beyond stratified semantics, the criteria for selecting the intended model become increasingly contrived. Furthermore, the assumption that the chosen criterion captures a fictitious programmer's natural reasoning process rests on shallow ground.

In contrast, the procedural, forward chaining approach provides semantics that is extremely simple to describe. However, the semantics is purely computational and does not yield a solution easily justifiable in model theoretic terms, beyond the fact that the answer is one model of the program. Also, despite the simplicity of the computational semantics, programs can be hard to understand, as their effect is very sensitive to timing (e.g., see Examples 4.1 and 4.3). In terms of expressiveness, the Datalog-like languages with forward chaining semantics provide a unifying formalism capable of capturing the main classes of queries, including *fixpoint*, *while*, and all the way to computable queries. They can also incorporate in a natural manner nondeterminism and updates. Declarative semantics seems to hit an expressiveness ceiling with well-founded semantics, capturing the *fixpoint* queries.

The final test for the various semantics has to rest with programmers and adoption in practical languages. In that respect, the forward chaining approach was an early leader, having been adopted in production systems and expert systems [39, 59] as well as active databases [117], at a time when implementations of deductive databases were limited to prototypes (see [107] for an early survey of deductive database implementations). Subsequently, the landscape became much more diverse. There have been many implementations of Datalog variants, with declarative and procedural semantics (e.g., see [96]). Most practical Datalog-like languages use stratified negation, which emerges as the indisputable success story of declarative semantics. As seen above, Datalog-like languages with forward chaining semantics, with features including updates and nondeterminism, remain common in a limited class of applications, mostly those that can be viewed as data-driven reactive systems. Such applications include active databases, production systems, data-driven workflows, peer-to-peer data exchange, and systems supporting interactive features (e.g., see [10, 11, 46, 56, 77, 103, 117]).

Overall, the forward chaining paradigm has demonstrated its theoretical appeal and persistence in practice. It will most likely continue to provide a useful alternative within the rich landscape of Datalog.

ACKNOWLEDGMENTS

This work was supported in part by the ANR Headwork project ANR-16-CE23-0015 and the National Science Foundation under award III-1815247. The author is grateful to Pierre Bourhis and Luc Segoufin for their insightful comments on this paper. Special thanks to Serge Abiteboul for revisiting our joint work and providing many useful suggestions for the presentation.

REFERENCES

[1] S. Abiteboul and G. Grahne. Update semantics for incomplete databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 1–12, 1985.
[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison Wesley, 1995.
[3] S. Abiteboul, E. Simon, and V. Vianu. Non-deterministic languages to express deterministic transformations. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 218–229, 1990.
[4] S. Abiteboul and V. Vianu. A transaction language complete for database update and specification. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 260–268, 1987.
[5] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 240–250, 1988.
[6] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.
[7] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 209–219, 1991.
[8] S. Abiteboul and V. Vianu. Non-determinism in logic-based languages. *Annals of Math. and Artif. Int.*, 3:151–186, 1991.
[9] Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 358–367, 2005.
[10] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The Active XML project: an overview. *VLDB J.*, 17(5):1019–1040, 2008.
[11] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Émilien Antoine. A rule-based language for web data management. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 293–304, 2011.
[12] Serge Abiteboul and Paris C. Kanellakis. Object identity as a query language primitive. *J. ACM*, 45(5):798–842, 1998.
[13] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of active XML systems. *ACM Trans. Database Syst.*, 34(4):23:1–23:44, 2009. Also in PODS 2008.
[14] Serge Abiteboul and Victor Vianu. Fixpoint extensions of first-order logic and Datalog-like languages. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 71–79, 1989.
[15] Serge Abiteboul and Victor Vianu. Non-determinism in logic-based languages. *Ann. Math. Artif. Intell.*, 3(2-4):151–186, 1991.
[16] Serge Abiteboul and Victor Vianu. Collaborative data-driven workflows: think global, act local. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 91–102, 2013.
[17] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 110–117, 1979.
[18] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.
[19] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19,*

2010. Revised Selected Papers, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281. Springer, 2010.

[20] Mario Alviano and Andreas Pieris, editors. *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019*, volume 2368 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[21] Tom J. Ameloot. Declarative networking: Recent theoretical work on coordination, correctness, and declarative semantics. *SIGMOD Rec.*, 43(2):5–16, 2014.

[22] Tom J. Ameloot and Jan Van den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. In Alin Deutsch, editor, *Proc. of Intl. Conf. on Database Theory*, pages 86–98, 2012.

[23] Tom J. Ameloot, Jan Van den Bussche, William R. Marczak, Peter Alvaro, and Joseph M. Hellerstein. Putting logic-based distributed systems on stable grounds. *Theory Pract. Log. Program.*, 16(4):378–417, 2016.

[24] Tom J. Ameloot, Bas Ketsman, Frank Neven, and Daniel Zinn. Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the calm-conjecture. *ACM Trans. Database Syst.*, 40(4):21:1–21:45, 2016.

[25] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15:1–15:38, 2013. Also in PODS 2011.

[26] J. Anderson, M. Gaare, J. Holgun, N. Bailey, and T. Pratley. *The Datomic database*, pages 169–215. In Professional Clojure, Wiley Online Library, 2016.

[27] K. Apt and M. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982.

[28] K.R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, CA, 1988.

[29] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 1371–1382, 2015.

[30] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. Expressive languages for querying the semantic web. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 14–26, 2014.

[31] Pablo Barceló and Reinhard Pichler, editors. *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, volume 7494 of *Lecture Notes in Computer Science*. Springer, 2012.

[32] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 119–128, 2001.

[33] Luigi Bellomarini, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. Swift logic for big data and knowledge graphs. In *Proc. of the Intl. Joint Conference on Artificial Intelligence, IJCAI 2017*, pages 2–10, 2017.

[34] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. The Vadalog system: Datalog-based reasoning for knowledge graphs. *Proc. VLDB Endow.*, 11(9):975–987, 2018.

[35] Gerald Berger, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. The space-efficient core of Vadalog. In *Proc. ACM SIGMOD-SIGACT-SIGAI Symp. on Principles of Database Systems*, pages 270–284, 2019.

[36] N. Bidoit and C. Froidevaux. Minimalism subsumes default logic and circumscription. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 89–97, 1987.

[37] N. Bidoit and C. Froidevaux. General logic databases and programs: Default logic semantics and stratification. *J. Information and Computation*, 91(1):15–54, 1991.

[38] L. Brownston, R. Farrel, E. Kant, and N. Martin. *Programming Expert Systems in OPS5*. Addison Wesley, 1985.

[39] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley Longman Publishing Co., Inc., USA, 1985.

[40] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In Ronald Fagin, editor, *Proc. of Intl. Conf. on Database Theory*, volume 361, pages 14–30, 2009.

[41] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *Proc. of Intl. Conf. on Database Theory*, volume 361, pages 14–30, 2009.

[42] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 228–242, 2010.

[43] Andrea Calì and Michael Kifer. Containment of conjunctive object meta-queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 942–952, 2006.

[44] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Dl-lite: Tractable description logics for ontologies. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 602–607, 2005.

[45] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. Autom. Reason.*, 39(3):385–429, 2007.

[46] Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. Foundations of data-aware process analysis: a database theory perspective. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 1–12, 2013.

[47] A. K. Chandra. Programming primitives for database languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 50–62, 1981.

[48] A.K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[49] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25(1):99–128, 1982.

[50] A.K. Chandra and D. Harel. Horn clause queries and generalizations. *J. Logic Programming*, 2(1):1–15, 1985.

[51] E. F. Codd. A relational model of data for large shared data banks. *Comm. of the ACM*, 13(6):377–387, 1970.

[52] L. Corciulo, F. Giannotti, and D. Pedreschi. Datalog with non-deterministic choice computes NDB-PTIME. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 93.

[53] E. Dalhaus. Skolem normal forms concerning the least fixpoint. In E. Börger, editor, *Computation Theory and Logic*, volume 270, pages 101–106. Springer Verlag, Lecture Notes in Computer Science, Berlin/New York, 1987.

[54] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[55] Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors. *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*. Springer, 2011.

[56] Alin Deutsch, Richard Hull, and Victor Vianu. Automatic verification of database-centric systems. *SIGMOD Rec.*, 43(3):5–17, 2014.

[57] Jörg Flum, Max Kubierschky, and Bertram Ludäscher. Total and partial well-founded Datalog coincide. In *Proc. of Intl. Conf. on Database Theory*, pages 113–124, 1997.

[58] Jörg Flum, Max Kubierschky, and Bertram Ludäscher. Games and total Datalog¬ queries. *Theor. Comput. Sci.*, 239(2):257–276, 2000.

[59] C. L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, 1981.

[60] 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.

[61] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In *IEEE Symp. on Logic Programming*, pages 127–139, 1986.

[62] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 1–11, 1989.

[63] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 221–230, 1988.

[64] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:620–650, 1991.

[65] M. Gelfond and V. Lifschitz. The stable model semantics for logic programs. In *Intl. Conf. on Logic Programming*, pages 1070–1080, 1988.

[66] F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Nondeterminism in deductive databases. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 129–146, Los Altos, CA, 1991. Springer Verlag, Lecture Notes in Computer Science 566.

[67] Georg Gottlob and Christoph Koch. Monadic queries over tree-structured data. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 189–202, 2002.

[68] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004. Also in PODS 2002.

[69] Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog, and Sergio Flesca. The lixto data extraction project - back and forth between theory and practice. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 1–12, 2004.

[70] Georg Gottlob, Reinhard Pichler, and Emanuel Sallinger. Function symbols in tuple-generating dependencies: Expressive power and computability. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 65–77, 2015.

[71] Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 250–256, 2006.

[72] Georg Gottlob, Reinhard Pichler, and Fang Wei. Tractable database design through bounded treewidth. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 124–133, 2006.

[73] Georg Gottlob, Reinhard Pichler, and Fang Wei. Abduction with bounded treewidth: From theoretical tractability to practically efficient computation. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1541–1546, 2008.

[74] Georg Gottlob, Reinhard Pichler, and Fang Wei. Monadic datalog over finite structures of bounded treewidth. *ACM Trans. Comput. Log.*, 12(1):3:1–3:48, 2010. Also in PODS 2007.

[75] Georg Gottlob and Andreas Pieris. Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2999–3007, 2015.

[76] Sergio Greco, Domenico Saccà, and Carlo Zaniolo. Extending stratified datalog to capture complexity classes ranging from P to QH. *Acta Informatica*, 37(10):699–725, 2001.

[77] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Found. Trends Databases*, 5(2):105–195, 2013.

[78] Todd J. Green, Gregory Karvounarakis, Nicholas E. Taylor, Olivier Biton, Zachary G. Ives, and Val Tannen. ORCHESTRA: facilitating collaborative data sharing. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 1131–1133, 2007.

[79] Martin Grohe. The quest for a logic capturing PTIME. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 267–271, 2008.

[80] Joseph M. Hellerstein. Datalog redux: experience and conjecture. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 1–2. ACM, 2010.

[81] Joseph M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39(1):5–19, 2010.

[82] T. Imielinski and W. Lipski. The relational model of data and cylindric algebras. *Journal of Computer and System Sciences*, 28(1):80–102, 1984.

[83] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[84] Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando C. N. Pereira. The ORCHESTRA collaborative data sharing system. *SIGMOD Rec.*, 37(3):26–32, 2008.

[85] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.

[86] P. G. Kolaitis. The expressive power of stratified logic programs. *Information and Computation*, 90(1):50–66, 1991.

[87] P. G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint? In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 231–239, 1988.

[88] S. Konolige. On the relation between default and autoepistemic logic. *Artificial Intelligence*, 35(3):343–382, 1988.

[89] Nikolaos Konstantinou, Martin Koehler, Edward Abel, Cristina Civili, Bernd Neumayr, Emanuel Sallinger, Alvaro A. A. Fernandes, Georg Gottlob, John A. Keane, Leonid Libkin, and Norman W. Paton. The VADA architecture for cost-effective data wrangling. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 1599–1602, 2017.

[90] R. Krishnamurthy and S.A. Naqvi. Nondeterministic choice in datalog. In *5th Int'l. Conf. on Data and Knowledge Bases*, pages 416–424, Los Altos, CA, 1988. Morgan Kaufmann.

[91] Georg Lausen, Bertram Ludäscher, and Wolfgang May. On active deductive databases: The Statelog approach. In *Transactions and Change in Logic Databases, International Seminar on Logic Databases and the Meaning of Change, Schloss Dagstuhl, Germany, September 23-27, 1996 and ILPS '97 Post-Conference Workshop on (Trans)Actions and Change in Logic Programming and Deductive Databases, (DYNAMICS'97) Port Jefferson, NY, USA, October 17, 1997, Invited Surveys and Selected Papers*, volume 1472 of *Lecture Notes in Computer Science*, pages 69–106. Springer, 1998.

[92] V. Lifschitz. On the declarative semantics of logic programs with negation. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 177–192. Morgan Kaufmann, Los Altos, CA, 1988.

[93] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.

[94] Boon Thau Loo, Harjot Gill, Changbin Liu, Yun Mao, William R. Marczak, Micah Sherr, Anduo Wang, and Wenchao Zhou. Recent advances in declarative networking. In Claudio V. Russo and Neng-Fa Zhou, editors, *Practical Aspects of Declarative Languages - 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*, volume 7149 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[95] D. Maier and D. S. Warren. *Computing with Logic: Logic Programming with Prolog*. Benjamin Cummings, Menlo Park, CA, 1988.

[96] 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*, pages 3–100. ACM / Morgan & Claypool, 2018.

[97] R.C. Moore. Semantics considerations on non-monotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[98] Y.N. Moschovakis. *Elementary Induction on Abstract Structures*. North Holland, 1974.

[99] S. Naqvi and S. Tsur. *A language for data and knowledge bases*. Computer Science Press, Rockville, Maryland, 1989.

[100] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 37–48. ACM, 2012.

[101] C.P. Papadimitriou. A note on the expressive power of prolog. *Bulletin of the EATCS*, 26:21–23, 1985.

[102] The Laguna Beach Participants. Future directions in DBMS research. *SIGMOD Rec.*, 18(1):17–26, 1989.

[103] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Operational semantics for declarative networking. In *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009, Savannah, GA, USA, January 19-20, 2009. Proceedings*, volume 5418 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2009.

[104] Philippe Picouet and Victor Vianu. Semantics and expressiveness issues in active databases. *J. Comput. Syst. Sci.*, 57(3):325–355, 1998. Also in ICDT 1997.

[105] T. Przymusinski. Every logic program has a natural stratification and an iterated least fixpoint model. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 11–21, 1989.

[106] T. Przymusinski. Well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae*, XIII:445–463, 1990.

[107] R. Ramakrishnan and J.D. Ullman. A survey of deductive database systems. *J. Logic Programming*, 23(2):125–149, 1995.

[108] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1):80–132, 1980.

[109] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 205–217, 1990.

[110] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 1135–1149, 2016.

[111] Michael Stonebraker and Joseph M. Hellerstein, editors. *Readings in Database Systems, Third Edition*. Morgan Kaufmann, 1998.

[112] VADA project website. http://vada.org.uk, 2021. Accessed: 2021-03-04.

[113] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[114] M. Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.

[115] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. of Intl. Conf. on Database Theory*, pages 96–106, 2014.

[116] Victor Vianu. Rule-based languages. *Ann. Math. Artif. Intell.*, 19(1-2):215–259, 1997.

[117] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, 1995.

[118] 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.