Foundations and Trends® in Databases
Vol. 5, No. 2 (2012) 105–195
© 2013 T. J. Green, S. Huang, B. T. Loo, W. Zhou
DOI: 10.1561/1900000017

Datalog and Recursive Query Processing

Todd J. Green
LogicBlox Inc.
todd.green@logicblox.com

Shan Shan Huang
LogicBlox Inc.
ssh@logicblox.com

Boon Thau Loo
University of Pennsylvania
boonloo@cis.upenn.edu

Wenchao Zhou
Georgetown University
wzhou@cs.georgetown.edu

Contents

1 Introduction 106

1.1 Contributions and Roadmap 108

1.2 Relationship with Previous Surveys 109

1.3 First Example: All-Pairs Reachability 110

2 Language and Semantics 113

2.1 Language 113

2.2 Semantics 115

2.3 Negation 120

2.4 Aggregation 126

3 Recursive Query Processing 131

3.1 Bottom-up Evaluation 132

3.2 Top-down Evaluation 135

3.3 Magic Sets 141

4 Incremental Maintenance 151

4.1 Counting Algorithm for Non-recursive Queries 152

4.2 Delete and Re-Derive Algorithm (DRed) 153

4.3 Provenance-based Incremental Maintenance 155

4.4 Incremental Maintenance for Negation and Aggregates . . 155

5 Datalog Extensions **157**
5.1 Beyond Stratified Negation 157
5.2 Beyond Stratified Aggregation 159
5.3 Arithmetic and Infinite Relations 161
5.4 Functors 163
5.5 States and Updates 165

6 Applications **169**
6.1 Program Analysis 169
6.2 Declarative Networking 172
6.3 Data Integration and Exchange 175
6.4 Enterprise Software 180
6.5 Other Applications 183

Acknowledgements **186**

References **187**

Abstract

In recent years, we have witnessed a revival of the use of recursive queries in a variety of emerging application domains such as data integration and exchange, information extraction, networking, and program analysis. A popular language used for expressing these queries is Datalog. This paper surveys for a general audience the Datalog language, recursive query processing, and optimization techniques. This survey differs from prior surveys written in the eighties and nineties in its comprehensiveness of topics, its coverage of recent developments and applications, and its emphasis on features and techniques beyond "classical" Datalog which are vital for practical applications. Specifically, the topics covered include the core Datalog language and various extensions, semantics, query optimizations, magic-sets optimizations, incremental view maintenance, aggregates, negation, and types. We conclude the paper with a survey of recent systems and applications that use Datalog and recursive queries.

T. J. Green, S. Huang, B. T. Loo, W. Zhou. *Datalog and Recursive Query Processing*. Foundations and Trends® in Databases, vol. 5, no. 2, pp. 105–195, 2012.

DOI: 10.1561/1900000017.

1

Introduction

Mainstream interest in Datalog in the database systems community flourished in the eighties and early nineties. During this period, there were several pioneering Datalog systems, primarily from academia. Two of the more prominent ones with complete implementations include Coral [99] and LDL++ [20]. Some ideas from these early research prototypes made it into mainstream commercial databases. For instance, Oracle, DB2, and SQL Server provide support for limited forms of support for recursion, based on SQL-99 standards. However, a perceived lack of compelling applications at the time [113] ultimately forced Datalog research into a long dormancy, and stifled its use in practice. Coral and LDL++ ceased active development in 1997 and 2000 respectively, and commercial systems did not extend a limited form of Datalog.

In recent years, however, Datalog has reemerged at the center of a wide range of new applications, including data integration [68, 43, 50], declarative networking [80, 77, 75], program analysis [29], information extraction [110], network monitoring [10], security [85, 60], optimizations [73], and cloud computing [15, 16]. Compared to the state-of-the-art of two decades ago, the modern systems that drives these emerging applications have significantly more mature and complete Datalog implementations, and often times deploy applications that are orders of magnitude larger in code size and complexity compared to the older generation of Datalog programs.

In terms of modern academic systems, the IRIS reasoner [59] is an open-source general purpose Datalog execution engine with support for optimizations, stratified and locally stratified negation. There are also publicly available Datalog systems tailored for specific applications. These include the Orchestra system for collaborative data sharing [92], BDDBDD [24] for program analysis, the RapidNet [101] declarative networking platforms, and the Bloom [16] platform for declarative programming in the cloud.

In the commercial world, a major development is the emergence of enterprise Datalog systems, most notably LogicBlox [4], Datomic [2], Semmle [7], and Lixto [49]. Semmle and Lixto are targeted at specific domains of program analysis and information extraction respectively, while LogicBlox and Datomic aim to provide a general platform for developing enterprise software.

The revival of Datalog in the new generation of applications is driven by the increasing need for high-level abstractions for reasoning about and rapidly developing complex systems that process large amounts of data, and are sometimes distributed and parallel. Datalog provides a declarative interface that allows the programmer to focus on the tasks (“what”), not the low-level details (“how”). A common thread across these systems is the use of the Datalog language as a declarative abstraction for querying graphs and relational structures, and implementing iterations and recursions. Its clear and simple syntax with well understood semantics aims to achieve the best of both worlds – having a rich enough language to support a wide range of applications, yet at a high and concise level that makes rapid prototyping easy for programmers without having to worry about low level messy details related to robustness and parallelism. The high-level specifications also make code analysis easier, for applying optimizations and for reasoning about transactions and safety.

1.1 Contributions and Roadmap

This survey paper aims to provide an accessible and gentle introduction to Datalog and recursive query processing to readers with some basic background in databases (in particular, SQL and the relational model). Given the wide range of research literature on Datalog spanning decades, we identify a "practical" subset of Datalog based on recent advances in the adoption of Datalog. In particular, our survey aims to cover the following:

- **Language.** Core Datalog syntax and semantics. (Chapter 2)
- **Query processing.** Recursive query processing techniques for executing Datalog programs efficiently, using the bottom-up and top-down evaluation strategies, such as the well-known *semi-naïve* [22, 21] and *Query/Subquery (QSQ)* [67] evaluation strategies. (Chapter 3)
- **Incremental maintenance.** Extensions to query processing techniques in the previous chapter, to include mechanisms for incrementally updating the materialized views of a Datalog program, as the input data changes, without having to recompute the entire Datalog program from scratch. (Chapter 4)
- **Common extensions.** Each application domain takes the core Datalog language and then further customizes and extends the core language and implementation techniques to meet its particular needs. Here, we discuss extensions to incorporate negation, aggregation, arithmetic, uninterpreted functions, and updates, as well as the query processing techniques to handle these extensions. (Chapter 5).

The survey concludes in Chapter 6 with a brief survey of recent applications of Datalog, in the domains of program analysis, declarative networking, data integration and exchange, enterprise software systems, etc.

1.2 Relationship with Previous Surveys

Our survey serves as an entry point into several other survey papers and books on Datalog. We briefly mention some of them:

- Bancilhon et al. [23] surveys and compares various strategies for processing and optimizing recursive queries in a greater depth compared to our survey.

- Ceri et al. [32] presents the syntax and semantics of Datalog along with evaluation and optimization techniques for efficient execution. Extensions to the Datalog language, such as built-in predicates and negation are also discussed.

- Ramakrishnan and Ullman [100] provides a high-level overview of the Datalog language, query evaluation and optimizations, and more advanced topics on negation and aggregation in a few pages. This should be viewed as a "quick-starter" guide for someone exposed to Datalog for the first time.

- Textbooks [12, 27, 33, 118, 36] cover some topics (e.g. language, semantics, magic sets) in greater detail than our survey. Abiteboul et al. [12] in particular is a widely used textbook geared towards a database theory audience.

Overall, our survey is broader than Bancilhon [23], which focuses primarily on query processing, and Ramakrishnan and Ullman [100], which surveys Datalog systems (which are now more than a decade old) with a brief discussion on query processing and optimizations. We cover a breath of topics similar to the surveys [32, 88], but provide significantly more details on systems issues related to query processing, incremental maintenance, and modern applications. Compared to all of the above surveys, we provide a more systems approach in presentation of classical topics, and discuss only extensions relevant to modern applications.

1.3 First Example: All-Pairs Reachability

We begin with a high level introduction to the Datalog language and its basic evaluation strategy. As our first example, we consider a Datalog program that computes *all-pairs reachability*, essentially a transitive closure computation in a graph for figuring out all pairs of nodes that are connected (reachable) to each other.

```
r1 reachable(X,Y) :- link(X,Y).
r2 reachable(X,Y) :- link(X,Z), reachable(Z,Y).
query(X,Y) :- reachable(X,Y).
```

The above two rules, named as *r1* and r2, derive the `reachable` nodes (i.e. `reachable(X,Y)` using facts about directly linked nodes (i.e. `link(X,Y)`). Here, we use capital letters `X` and `Y` to signify that they are variables in the domain of all the nodes. The output of interest in this program, as denoted by the special predicate `query(X,Y)`, is the set of derived `reachable` facts. The input graph in this case can represent a network of routers, and forms a basis for implementing network routing protocols [80], web crawlers [81], and network crawlers [79].

Rule *r1* expresses that node `X` is reachable from `Y` (i.e. `reachable(X,Y)`) if they are directly linked. Rule *r2* is a bit more interesting, as it specifies the `reachable` relation in terms of itself: `(X,Y)` are reachable from one another if `X` has a direct link to a node (`Z`) that is reachable to `Y`. We refer to rules such as *r2* as *recursive* rules, since the *reachable* relation appears in both the rule body (right of " `:-` ") and head (left of " `:-` "). Rule *r2* is also a *linear* recursive rule ,since `reachable` appears only once in the rule body.

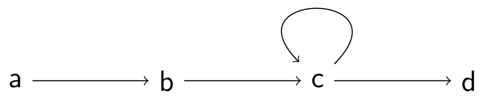


Figure 1.1: Example graph used for reachability computation.

We illustrate the execution of Datalog rules by evaluating the `reachable` rules over the graph shown in Figure 1.1, which depicts a network consisting of three nodes and four direct `link`s. Thus, there are

(initial base tuples)
`link`

X	Y
a	b
b	c
c	c
c	d

(iteration 1)
`reachable`

X	Y
a	b
b	c
c	c
c	d

(iteration 2)
`reachable`

X	Y
a	b
b	c
c	c
c	d
a	c
b	c
b	d
c	d

(iteration 3)
`reachable`

X	Y
a	b
b	c
c	c
c	d
a	c
b	d
a	d

Figure 1.2: Tuples derived by the *All-pairs Reachability* program for each iteration. New tuples derived in the current iteration that are not known in prior iterations are shaded.

four initial entries (tuples) in `link`: `link(a,b)`, `link(b,c)`, `link(c,c)`, and `link(c,d)`.

Intuitively, rule evaluation can be understood as the repeated application of rules over existing tuples to derive new tuples, until no more new tuples can be derived (i.e. evaluation has reached a *fixpoint*). Each application of rules over existing tuples is referred to as an *iteration*. This evaluation strategy is often times referred to as the *naïve* evaluation strategy.

The evaluation of the reachability rules over the network in Figure 1.1 reaches a fixpoint in three iterations, as shown in Figure 1.2. In iteration 1, rule *r1* takes as input the initial `link` tuples, and use that to generate 4 `reachable` tuples. These tuples essentially represent all pairs of nodes reachable within one hop. In the next two iterations, all `reachable` tuples generated in previous iterations are used as input to rule *r2* to generate more `reachable` tuples that are two and three hops apart. Iteration 4 (not shown in the figure) derives the same set of tuples as iteration 3, and hence, a fixpoint is reached. Given that no two nodes are separated by more than 3 hops, the recursive query completes in 4 iterations.

As an optimization, instead of using all derived facts as input to rules at each iteration, one can suppress the evaluation that uses *only* tuples already learned in prior iterations when computing new tuples the next iteration. For instance, when generating new facts in iteration 3, rule *r2* will not evaluate for inputs `reachable(b,c)` and `link(a,b)`, since they have already been used in iteration 1. The intuitive description above corresponds loosely to the *semi-naïve* evaluation strategy, which will be described in greater detail in Chapter 3.

Note that the above approach is a *bottom-up* evaluation technique, where existing facts are used as input to rule bodies to derive new facts. A fixpoint is reached when no new facts are derived. This is also known as a *forward-chaining* style of evaluation. An alternative approach used in Prolog [112] uses a goal-oriented *backward-chaining* approach, starting from the goal (i.e. query), and then expanding the rule bodies in a top-down fashion.

A top-down approach allows for an evaluation strategy that focuses only on facts necessary for the goal. However, a bottom-up evaluation approach used in Datalog allows us to draw upon a wealth of query processing and optimization techniques to draw upon for doing the computations efficiently even when datasets are too large to fit in main memory. Moreover, as we show in Section 3.3, query optimization techniques can optimize Datalog programs for bottom-up evaluation, to avoid deriving facts not relevant to answering queries.

2

Language and Semantics

We present the formal syntax and semantics of Datalog in this chapter. We restrict our presentation here to *core* Datalog, and include a discussion on straightforward forms of negation with clear semantics. We defer extensions (such as advanced negation and aggregation, arithmetic, and functor) to Chapter 5.

2.1 Language

Datalog program. A Datalog *program* is a collection of Datalog *rules*, each of which is of the form:

$$A \texttt{ :- } B_1, B_2, \ldots, B_n\,.$$

where $n \geq 0$, A is the *head* of the rule, and the conjunction of $B_1, B_2, \ldots, B_n$ is the *body* of the rule. The rule can be read informally as "B_1 and B_2 and ... and B_n implies A". In a Datalog program, we adopt the logic programming tradition in using lowercase for constants and predicate symbols and uppercase for variables.

Terminology. We provide some basic terminology used throughout the paper to describe Datalog programs of the form shown above. A *term* in Datalog is a constant or a variable. An *atom* (or *goal*) is a predicate symbol (function) along with a list of terms as arguments.[1] For example, in the above Datalog rule, A, B_1, B_2, ..., and B_n stand for atoms. An atom containing only constant arguments is called a *ground atom* (atom).

We further divide predicate symbols into two categories: *extensional database predicates* (*EDB predicates*), which correspond to the source tables of the database, and *intensional database predicates* (*IDB predicates*), which correspond to derived tables computed by Datalog programs. Entries in the source and derived tables are often referred to as *base* and *derived* atoms respectively. In our presentation, we also include in the core language the *built-in comparison predicates* $=, \neq, <, >, \leq, \geq$ which can be used to compare domain values.[2]

A *database instance* is a set of ground atoms. We use $I, J, K, \ldots$ to denote database instances. A *source database instance* is a database instance containing only EDB atoms. The *active domain* of a database instance is the set of all constants that occur in the database. We limit our analysis to core Datalog that considers only finite database instances (i.e., database instances whose active domains are finite).

Example 2.1. Refer to the all-pairs reachability program P from Section 1.3. `link` is an EDB predicate, while `reachable` is an IDB predicate. The distinguished query predicate refers to the *query* rule, which indicates that `reachable` predicate is the output of interest. Each atom contains a number of arguments, e.g. the `link(X,Y)` atom has arguments `X` and `Y` with an arity of 2. The source database instance in this case refers to the initial `link` base table shown in Figure 1.2. The derived table in this case is `reachable`.

The `link` table and the `reachable` table compromise a database instance, where `link` table is the source database instance. The active

[1]Note that "term" and "atom" are used differently here than they are in Prolog.

[2]The core Datalog presented in more theoretical treatments of the subject [12] usually does not assume an ordered domain or include built-in predicates, but for practical applications, these features are taken for granted.

domain corresponds to the set of nodes in the network (i.e., $\{a, b, c, d\}$ in the example).

A Datalog rule must also satisfy the *range restriction property*, which says that every variable occurring in the head of a rule must also occur in a predicate atom in the body, or be equivalent (modulo the comparisons in the body) to a variable occurring in a predicate atom in the body. Note that a rule may have an empty body (when $n = 0$). In this case the body is considered unconditionally `true`, the head must be ground (to satisfy range restriction), and we call such a rule a *fact rule*.

Given a Datalog program with a collection P of rules, the *Herbrand base* of P, denoted $B(P)$, is the set of all ground atoms that can be formed from the (EDB or IDB) predicates in P. A *ground instance* of a rule is obtained by substituting all variables of the rule with constants.

2.2 Semantics

An appealing aspect of core Datalog is that its semantics can be defined in three rather different but, as it turns out, equivalent ways. We present an informal sketch of the three semantics in this section, emphasizing intuitions and deferring most formal details to the references.

2.2.1 Model-theoretic Semantics

The most purely "declarative" semantics of core Datalog is based on the standard model-theoretic semantics of first-order logic. In this semantics, we view Datalog rules as logical *constraints*. For example, considering the all-pairs reachability example from Section 1.3, the two rules correspond to the logical constraints

$$(\forall X \forall Y)(\texttt{link}(X, Y) \rightarrow \texttt{reachable}(X, Y))$$
$$(\forall X \forall Y \forall Z)(\texttt{link}(X, Y) \wedge \texttt{reachable}(Y, Z) \rightarrow \texttt{reachable}(X, Z))$$

Given a source database instance I and Datalog program P, a *model* of P is a database instance I' that *extends* I (i.e., $I \subseteq I'$) and satisfies all the rules of P viewed as constraints. A given source instance and program will in general have many models; however, among all the models, it turns there is always one which is smallest.

Theorem 2.1 ([119])**.** For any core Datalog program P and source database instance I, there exists a *minimal model* I' of P extending I. That is, I' is a submodel of any other model I'' of P extending I. Moreover, I' is polynomial in the size of I (for a fixed program P).

Proof. (Sketch.) The theorem follows immediately once one has established two simple facts: (1) every Datalog program P and source database instance I has at least *one* model—namely, the instance I' obtained by adding to I all atoms over the active domain of I—and this model is, moreover, polynomial in the size of I (in fact, the exponent of the polynomial is the maximum arity of any predicate symbols in P); and (2) the intersection of two models is again a model. □

According to the model-theoretic semantics, this minimal model is the one that should be computed, and we denote it by $P(I)$. It captures our intuition that the result of evaluating a Datalog program should satisfy all the rules of the program, while not having more atoms in it than necessary.

Example 2.2. Refer to the all-pairs reachability program P from Section 1.3, and suppose the database instance I contains the base facts shown in Figure 1.2. Then P has infinitely many models containing I; two examples are:

$I_1 =$

`link`

X	Y
a	b
b	c
c	c
c	d

,

`reachable`

X	Y
a	b
b	c
c	c
c	d
a	c
b	d
a	d

$I_2 =$

`link`

X	Y
a	b
b	c
c	c
c	d
b	f

,

`reachable`

X	Y
a	b
b	c
c	c
c	d
a	c
b	d
a	d
b	f
a	f
g	h

Note that I_1 is a subinstance of I_2, which also contains I and satisfies the rules of P, but contains some "extraneous" atoms (highlighted in shade). In fact, I_1 is the smallest model of P containing I.

2.2.2 Fixpoint-theoretic Semantics

The model-theoretic semantics of core Datalog gives a clear definition of *what* a Datalog program should compute, but does not explain *how* it might be computed. This question is given a more satisfactory answer by the *least fixpoint* semantics. This turns out to be equivalent to the minimal model semantics [119], but lends itself more directly to practical, bottom-up evaluation strategies (such as those to be described in Section 3).

The least fixpoint semantics is based on the *immediate consequence operator* T_P for a Datalog program P, which maps database instances to database instances. An atom A is an *immediate consequence* of a Datalog program P and database instance I if A is a ground EDB atom in I, or A `:-` $B_1, \ldots, B_n$ is a ground instance of a rule in P, and $B_1, \ldots, B_n$ are in I. Then we define T_P as follows: $A \in T_P(I)$ iff A is an immediate consequence of P and I.

Example 2.3 (ht)**.** For example, if I is the instance shown below (which contains the ground atoms of the `link` EDB relation and an empty `reachable` IDB relation), and P is the transitive closure program, then applying the immediate consequence operator T_P for P to I yields the instance $T_P(I)$ shown below:

$I =$

`link`

X	Y
a	b
b	c
c	c
c	d

,

`reachable`

X	Y

$T_P(I) =$

`link`

X	Y
a	b
b	c
c	c
c	d

,

`reachable`

X	Y
a	b
b	c
c	c
c	d

Observe that `reachable` in $T_P(I)$ contains only atoms derived using the first rule of the program, since `reachable` is empty in I hence the second rule of the program does not (yet) produce any new atoms.

Note that the immediate consequence operator is *monotone*: whenever $I \subseteq J$, then $T_P(I) \subseteq T_P(J)$. A *fixpoint* for T_P is a database instance I such that $T_P(I) = I$. In general, there may be many fixpoints for T_P. A *least fixpoint* for T_P containing I is a fixpoint that is a subset of any other fixpoint for T_P containing I. Intuitively, a least fixpoint does not contain any “extraneous” atoms. One can show that such a fixpoint always exists and agrees with the model-theoretic semantics:

Theorem 2.2. For any core Datalog program P and source instance I, a least fixpoint of T_P containing I exists and is equal to $P(I)$.

The least fixpoint semantics leads to a constructive procedure for evaluating Datalog programs. The idea is to repeatedly apply the immediate consequence operator, starting with the source database instance, until reaching a fixpoint. One can show that this process yields precisely the least fixpoint, and moreover, it will be reached in a number of steps that is polynomial in the size of the source database instance. Since each application of T_P can be computed in polynomial time in the size of its input, this means that the least fixpoint can also be computed in polynomial time.

Example 2.4. Refer again to the all-pairs reachability example from Section 1.3, and assume again that the database instance I initially contains the atoms shown in Figure 1.2. Then repeatedly applying the immediate consequence operator yields the following database instances:

$T_P(I) =$

`link(a,b)`	`reachable(a,b)`
`link(b,c)`	`reachable(b,c)`
`link(c,c)`	`reachable(c,c)`
`link(c,d)`	`reachable(c,d)`

$T_P^2(I) =$

`link(a,b)`	`reachable(a,b)`	`reachable(a,c)`
`link(b,c)`	`reachable(b,c)`	`reachable(b,d)`
`link(c,c)`	`reachable(c,c)`	
`link(c,d)`	`reachable(c,d)`	

$T_P^3(I) =$

link(a,b)	reachable(a,b)	reachable(a,c)
link(b,c)	reachable(b,c)	reachable(b,d)
link(c,c)	reachable(c,c)	reachable(a,d)
link(c,d)	reachable(c,d)	

$T_P^4(I) =$

link(a,b)	reachable(a,b)	reachable(a,c)
link(b,c)	reachable(b,c)	reachable(b,d)
link(c,c)	reachable(c,c)	reachable(a,d)
link(c,d)	reachable(c,d)	

Note that the last two instances are identical, i.e., $T_P^3(I) = T_P^4(I)$, so in this example we reach the least fixpoint after three iterations. Note also that the least fixpoint is the same as the minimal model I_1 from Example 2.2.

2.2.3 Proof-theoretic Semantics

We close this section with a sketch of the third semantics for core Datalog, which defines the result of a Datalog program to be precisely the set of atoms that can be *proved* using the source database instance and the rules of the program. For example, considering the same example as above, a proof of the atom `reachable(a,d)` is as follows:

	Atom	**Reason**
1	`link(c,d)`	given
2	`reachable(c,d)`	using (1) and the first rule
3	`link(b,c)`	given
4	`reachable(b,d)`	using (2,3) and the second rule
5	`link(a,b)`	given
6	`reachable(a,d)`	using (4,5) and the second rule

A proof of an output atom can also be represented graphically as a *proof tree*, such as the one shown in Figure 2.1. In a proof tree, leaf nodes correspond to source atoms in the EDB instance, while internal nodes correspond to derived IDB atoms and are labeled with the rule used to derive them from the child atoms.

Note that the height of the proof tree for `reachable(a,d)` is 3, which corresponds to the fact that the `reachable(a,d)` is derived in the third iteration of applying the immediate consequence operator.

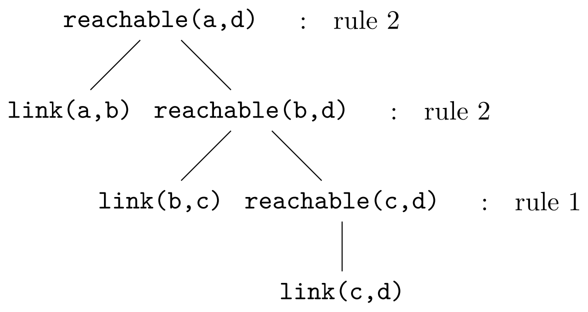


Figure 2.1: A proof tree

Theorem 2.3. The set of atoms provable from a source database instance I using the rules of P is precisely $P(I)$.

There are two basic strategies for generating proofs of output atoms. In the *bottom-up* (*backward chaining*) approach, we view rules as "factories" (to borrow an apt phrase from [12]), and apply them repeatedly to derive more and more proofs of atoms. In the *top-down* (*forward chaining*) approach, we start with a candidate output atom (the *goal*) and attempt to construct a proof of it, by recursively attempting to prove other atoms from which the goal can be derived. Top-down strategies are traditionally used in the evaluation of Prolog programs, but for Datalog, bottom-up evaluation has traditionally been preferred. We discuss both styles of evaluation further in Chapter 3.

2.3 Negation

We have seen that core Datalog programs are *monotone*: if $I \subseteq J$, then $P(I) \subseteq P(J)$. This means that core Datalog is incapable of expressing non-monotone queries, for example, the query which retrieves all *unreachable* pairs of nodes in a graph.[3] Since this is a major practical shortcoming, we consider here extending core Datalog to incorporate the use of *negation* in rule bodies (but not in rule heads). We denote the resulting language by Datalog$^{\neg}$.

[3]In fact, it is known that there are even monotone, polynomial-time computation not expressible in core Datalog [14].

Example 2.5. For example, to compute all pairs of disconnected nodes in a graph, we would write

```
reachable(X,Y) :- link(X,Y).
reachable(X,Y) :- link(X,Z), reachable(Z,Y).
node(X) :- link(X,Y).
node(Y) :- link(X,Y).
unreachable(X,Y) :- node(X), node(Y), not reachable(X,Y).
```

While the intuitive meaning of the program in the example is clear, a systematic treatment of Datalog$^\neg$ turns out to be trickier than one might first expect, as the use of recursion through negation leads to programs whose expected meaning is unclear.

Example 2.6. Consider the following Datalog$^\neg$ program:

```
p :- not q.
q :- not p.
```

This program has two minimal models, $I_1 = \{\mathtt{p}\}$ and $I_2 = \{\mathtt{q}\}$, but no unique minimal model. Also, each of these models is also a least fixpoint, but there is no unique least fixpoint. Finally, following a proof-theoretic approach, it is not clear how to prove either `p` or `q` (since neither atom is logically entailed by the rules), so it seems that neither atom should be in the output of the program. However, a database instance containing neither atom fails to satisfy the rules of the program viewed as logical constraints.

In light of examples such as the one above, many semantics for Datalog$^\neg$ have been proposed over the years. We present here the most straightforward and natural of these, based on a syntactic restriction called *stratified negation* which disallows recursion through negation. We present stratified negation in two steps: first, we describe the simpler *semipositive Datalog*$^\neg$, then we extend the idea to *stratified Datalog*$^\neg$.

2.3.1 Semipositive Datalog

The idea in semipositive Datalog$^\neg$ is that, syntactically, negation appears only in rule bodies (as before) and furthermore only on EDB predicates (but not on IDB predicates). This allows us to express the set difference operation from relational algebra, for example, as well as other queries of interest.

Example 2.7. Continuing with our graph example, we might be interested in computing pairs of nodes which are not directly linked, but are still reachable from one another. This can be expressed in semipositive Datalog$^\neg$ by adding a rule to the transitive closure program from Chapter 1.3:

```
reachable(X,Y) :- link(X,Y).
reachable(X,Y) :- link(X,Z), reachable(Z,Y).
indirect(X,Y) :- reachable(X,Y), not link(X,Y).
```

We require also a *safety* condition which says that every variable in the body of a rule must occur in at least one *positive* (i.e., not negated) atom, as in the example above. This is required to ensure that the results of programs are finite and that their results depend only on the actual contents of the database. (In database theory, this is formalized as the notion of *domain independence* [12].)

It is straightforward to extend the semantics of core Datalog to semipositive Datalog$^\neg$, because we can think of a negated EDB predicate `not p` as just another EDB predicate $\overline{\texttt{p}}$ (where the complement is taken over the active domain of the EDB relations). Using the same definition of model and immediate consequence operator T_P as the core Datalog, we have the following the theorem.

Theorem 2.4. Let P be a semipositive Datalog$^\neg$ program. Then for any EDB database instance I:

(1) P has a unique minimal model J such that J agrees with I on the EDB predicates.

(2) T_P has a unique least fixpoint J such that J agrees with I on the EDB predicates.

(3) The minimum model and least fixpoints are identical and can be computed in polynomial time by applying T_P repeatedly to I.

We denote by $P(I)$ the semantics of a semipositive Datalog$^\neg$ program according to either of the equivalent formulations above. Note, however, the proof-theoretic model discussed in Section 2.2.3 does not straightforwardly apply here: the question of whether there exists a proof tree becomes complicated, involving the need to show the non-existence of a proof subtree. As for core Datalog, evaluation of semi-positive Datalog$^\neg$ programs can be done in polynomial time in the size of the database instance.

2.3.2 Stratified Negation

The Datalog$^\neg$ program in Example 2.5 is not semipositive, because a negated IDB predicate (`not reachable`) appears in the fifth rule. Nevertheless, the desired semantics of the program seems clear. Conceptually, we want to proceed in two steps: first, compute the `reachable` and `node` relations using the first four rules of the program; then, "freeze" the contents of those IDB relations—viewing them now as EDB relations, throwing out the first four rules—and compute `unreachable` with the fifth rule, using the semipositive Datalog$^\neg$ semantics. This is the basic idea behind *stratified Datalog*$^\neg$.

In the stratified semantics, we consider Datalog$^\neg$ programs that can be written as a *sequence* $P_1, \ldots, P_n$ of semipositive Datalog programs. At each step in the sequence, once we have computed the result of that step, the IDB predicates in the step become EDB predicates ("materialized views") for the next step.

Definition 2.1. A *stratification* of a Datalog$^\neg$ program P is an ordered partition of the IDB predicates in P into *strata* $P_1, \ldots, P_n$ such that

(1) If A `:-` $\ldots, B, \ldots$ is a rule in P, and A is in stratum P_i while B is in stratum P_j, then $i \geq j$;

(2) If A `:-` $\ldots,$`not` $B, \ldots$ is a rule in P, and A is in stratum P_i while B is in stratum P_j, then $i > j$.

A Datalog$^\neg$ program that admits a stratification is called *stratifiable*. For instance, the program in Example 2.5 is stratifiable, but the program in Example 2.6 is not. We shall discuss a simple procedure for checking whether a program is stratifiable in a moment.

Given a stratifiable Datalog$^\neg$ program P, the semantics is given by a two-step procedure:

(1) Compute a stratification $P_1, \ldots, P_n$ of P;

(2) Evaluate $P_1, \ldots, P_n$ in sequence as semipositive Datalog$^\neg$ programs, where the IDB predicates of P_i are considered as EDB predicates for P_j, $j > i$.

A Datalog$^\neg$ program can have many stratifications, so the question arises whether the above (non-deterministic) procedure defines the semantics uniquely. However, it turns out that the particular choice of a stratification does not matter:

Theorem 2.5 ([19]). If $P_1, \ldots, P_n$ and $P'_1, \ldots, P'_m$ are stratifications of the same Datalog$^\neg$ program P, then for any source database instance I, evaluating $P_1, \ldots, P_n$ and $P'_1, \ldots, P'_m$ on I yields the same result.

We can therefore talk about *the* result of evaluating P on I under the stratified semantics, which we denote by $P(I)$.

It remains to discuss how to compute a stratification of a Datalog$^\neg$ program P. This can be done via the device of a finite *precedence graph* G_P for P. The vertices in G_P are the IDB predicates of P, and the edges in G_P are as follows:

- If A `:-` $\ldots B \ldots$ is a rule in P, then (B, A) is an edge in G_P;
- If A `:-` $\ldots$`not` $B \ldots$ is a rule in P, then (B, A) is an edge in G_P labeled $\neg$.

We can use the precedence graph to check for stratifiability:

Proposition 2.1. A Datalog$^\neg$ program P is stratifiable iff its precedence graph G_P has no cycle containing an edge labeled $\neg$.

We can also use the precedence graph to compute a stratification for P, via the following procedure:

(1) Compute the strongly connected components of G_P, and let these define the strata for the program;

(2) Perform a topological sort of the strongly connected components to determine an ordering of the strata.

Usually these two steps are performed by a single combined algorithm (which was first introduced by Kosaraju in an unpublished paper), and the time complexity of the algorithm is linear to the number of predicates in the program.

Example 2.8. Consider the Datalog$^\neg$ program P from Example 2.5. The precedence graph G_P for this program is as follows (`link` is not shown as a parent of `reachable` since it is an EDB):

$$\begin{tikzcd}
\texttt{node} \arrow[r] & \texttt{unreachable} & \texttt{reachable} \arrow[l, "\neg"'] \arrow[loop right]
\end{tikzcd}$$

Note that although the graph has a cycle, the cycle does not go through the edge labeled $\neg$, so the program is stratifiable. Observe that the graph has three strongly connected components, each containing exactly one vertex. Sorting them topologically yields, for example, the stratification: (1) `reachable`; (2) `node`; (3) `unreachable`.

Next, consider the instance of `link` from Figure 1.2. After round (1) of evaluation, we compute `reachable` as shown in Examples 2.2 and 2.4. In round (2) we compute `node`, and in round (3) we compute `unreachable` using the instances of `reachable` and `node` from (1) and (2) as EDB relations; the results include the following atoms:

`node(a)`	`unreachable(a,a)`	`unreachable(c,b)`	`unreachable(d,d)`
`node(b)`	`unreachable(b,a)`	`unreachable(d,a)`	
`node(c)`	`unreachable(b,b)`	`unreachable(d,b)`	
`node(d)`	`unreachable(c,a)`	`unreachable(d,c)`	

Once again, query evaluation of stratifiable Datalog$^\neg$ programs can be done in time polynomial in the size of the source database instance.

Remarks on expressiveness. We have seen that stratified Datalog$^\neg$ strictly extends core Datalog in expressiveness by allowing one to express non-monotonic queries such as non-reachability, while at the same time preserving guarantees of termination in polynomial time in the size of the database instance. A natural question to ask is, what is the precise expressiveness of stratified Datalog$^\neg$? A remarkable characterization of its expressiveness was provided independently for least fixpoint queries by Vardi [122], Immerman [58], and Livchak [74] (with an explicit statement for Datalog$^\neg$ given by Papadimitriou [94]):

Theorem 2.6 (Immerman-Vardi)**.** Stratified Datalog$^\neg$ *captures* the polynomial time queries on ordered databases. That is, any query that can be computed in polynomial time on an ordered database can be expressed by a stratified Datalog$^\neg$ program.

Thus, stratified Datalog$^\neg$ expresses *exactly* those queries which are tractable from a theoretical perspective. Several remarks are in order though. First of all, "queries" in the above is used in a precise and somewhat narrow sense, and refers to a class of database transformations satisfying certain requirements, in particular *genericity* [12]. This rules out certain kinds of functions, in particular those involving commonly-used *aggregates* like `sum` or `count`, which can be computed in polynomial time and are certainly considered "queries" in practice, but rely on information (such as arithmetical functions) not explicitly encoded as tables in the database. (We discuss extensions to incorporate arithmetic in Section 5.3 and aggregates in Section 2.4.) Second, the result depends crucially on the assumption that queries have access to an order predicate. In fact, it has been shown that, in the presence of order on all the constants in the active domain, semipositive Datalog has the same expressive power as stratified Datalog$^\neg$. The question of the existence of a query language capturing the polynomial time queries on *unordered* databases is a longstanding open question in database theory [72].

2.4 Aggregation

Many applications require the computation of various kinds of summary information over the database. For example, rather than returning all pairs of connected nodes in a graph, one might be interested in computing a summary giving the *count* of nodes reachable from a node. This is an example of a query involving an *aggregate function*, in this case, the `count` function. As we shall see, the query can be expressed in Datalog extended with `count` as follows:

```
reachable(X,Y) :- link(X,Y).
reachable(X,Y) :- reachable(X,Z), link(Z,Y).
summary(X, count<Y>) :- reachable(X,Y).
```

Formally, an *aggregate function* is a mapping f from bags (multisets) of domain values to domain values.[4] Commonly-used aggregate functions in databases include `count`, `sum`, `max`, `min`, and `average`. More generally we fix a countable set $F = \{f_1, f_2, \dots\}$ of aggregate function symbols of various arities with associated aggregate functions. (For practicality, we assume that each such aggregate function is computable in polynomial time in the size of the input bag of domain values.) An *aggregate term* is an expression $f < t_1, \dots, t_k >$ where f is an aggregate function symbol of arity k and $t_1, \dots, t_k$ are (ordinary) terms.

Datalog rules with aggregates have the same syntax as presented in Section 2.1, but the head A may now contain both aggregate terms and (ordinary) terms. We require additionally that a rule satisfy an extended *range restriction property*:

- Every variable occurring in the head of a rule (including occurrences inside aggregate terms) must also occur in the body.
- No variable occurring in an aggregate term in the head of a rule may also occur in an ordinary term in the head of a rule.

The variables in A occurring only in ordinary terms are called the *grouping variables*. A *ground instance* of a rule with aggregate terms is obtained by substituting all grouping variables with constants.

We illustrate the syntax by comparison with SQL.

Example 2.9. Consider the following SQL aggregate query over a table `sales(product,city,amount)` recording product sales by city, which aggregates the sales by product:

[4]Alternatively, aggregates can be formalized without resorting to bags by following the approach of Klug [64].

```
select S.product, sum(S.amount)
from sales S
group by S.product;
```

In our syntax, this can be written as follows:

```
sales_by_product(Product, sum<Sales>) :- sales(Product, City, Sales).
```

Note that `Product` implicitly plays the role of the `group by` attribute in the Datalog rule. The Datalog syntax presented here does not contain an explicit `group by` construct.

The enhancement of Datalog with aggregate functions raises semantic difficulties reminiscent of those arising with negation, if recursion through aggregation is allowed. For instance, consider the program

```
p(X) :- q(X).
p(sum<X>) :- p(X).
```

evaluated on a source database containing facts `q(1)` and `q(2)`. Intuitively, what should be the result? By the first rule, we infer `p(1)` and `p(2)`; using the second rule, we infer `p(3)`. But then the second rule applies again, so it seems we must infer `p(6)`; then `p(12)`, `p(24)`, `p(48)`, and so on *ad infinitum*.

2.4.1 Stratified Aggregation

In order to safely rule out this and other such pathological cases, we restrict our attention here to cases where aggregation does not occur through recursion. In analogy with stratified negation, these are the so-called *aggregate stratified* [90] programs.

Definition 2.2. A *stratification* of a Datalog$^\neg$ with aggregates program P is an ordered partition of the IDB predicates into strata $P_1, \ldots, P_n$ such that conditions (1) and (2) of Definition 2.1 hold, along with

(3) If A `:-` $\ldots, B, \ldots$ is a rule in P such that A contains an aggregate term, and A is in stratum P_i while B is in stratum P_j, then $i > j$.

An aggregate stratification can be found, if one exists, via a straightforward extension of the precedence graph (Section 2.3.2) to record use of aggregates in addition to negation.

To define a fixpoint-theoretic semantics of aggregate stratified programs, we extend the immediate consequence operator to deal with aggregate functions as follows. Consider an instance I and a program P, and let r be a ground instance of a Datalog rule in P with aggregates of the form

$$R(t_1, \ldots, t_n) \texttt{ :- } B_1, \ldots, B_n\,.$$

Let Θ be the set of all substitutions θ of (non-grouping) variables of r by constants such that $\theta B_1, \ldots, \theta B_n$ are in I. Then a fact $R(c_1, \ldots, c_n)$ is an *immediate consequence* of the rule for instance I if (a) Θ is non-empty, and (b) the following conditions hold for $1 \leq i \leq n$:

- If term t_i is a constant, then $t_i = c_i$.
- If term t_i is an aggregate term $f(u_1, \ldots, u_k)$, we have

  $$c_i = f(\{(\theta u_1, \ldots, \theta u_k) \mid \theta \in \Theta\}),$$

  where the curly braces above indicate a bag rather than a set.

Then, we define T_P as before: $A \in T_P(I)$ iff A is an immediate consequence for some ground instance of a rule in P for I.

Finally, having extended the immediate consequence operator to work with aggregate-stratified programs, we define the fixpoint-theoretic semantics exactly as for stratified Datalog$^\neg$ (Section 2.3.2). Using the assumption that aggregate functions are polynomial time computable, it is easy to show that for any fixed query in this language, program evaluation can be done in time polynomial in the size of the database.

In addition to the fixpoint-theoretic semantics, one can give a natural model-theoretic semantics for aggregate stratified Datalog$^\neg$; see the paper by Mumick et al. [90] for details.

2.4.2 Aggregation Optimizations

Consider the following aggregate stratified Datalog program which computes the shortest paths in a network routing graph, along with their costs:

```
path(X,Y,[X,Y],C) :- link(X,Y,C).
path(X,Y,P,C) :- path(X,Z,P1,C1), link(Z,Y,C2), P = [P1,Y], C = C1 + C2.
shortest_path_len(X,Y,min<C>) :- path(X,Y,P,C).
shortest_path(X,Y,P,C) :- shortest_path_len(X,Y,C), path(X,Y,P,C).
```

A naïve execution of the program computes all possible paths, even those paths that do not contribute to the eventual shortest paths. Even worse, if the routing graph has cycles, then the set of all possible paths is infinite, and the fixpoint execution will not terminate.

These problems can be avoided with an optimization technique known as *aggregate selection* [115]. Intuitively, by applying aggregate selections to this program, each node only needs to propagate the current shortest paths for each destination to its neighbors. This propagation can be done whenever a shorter path is derived.

In brief, aggregate selection is performed via a syntactic rule transformation in the spirit of magic sets rewriting, along with a modified runtime evaluation strategy. For the shortest paths example, the rewriting replaces the `path` predicate in the rule body of the second recursive rule with `shortestPath`:

```
path_s1(X,Y,[X,Y],C) :- link(X,Y,C).
path_s1(X,Y,P,C) :- path_s1(X,Z,P1,C1), link(Z,Y,C2), P = [P1,Y], C=C1+C2.
shortest_path_len(X,Y,min<C>) :- path_s1(X,Y,P,C).
shortest_path(X,Y,P,C) :- shortest_path_len(X,Y,C), path_s1(X,Y,P,C).
Selections s1 = path_s1(X,Y,P,C) : groupby(path_s1(X,Y,P,C),[X,Y],min(C))
```

The predicate `path_s1` denotes the original `path` predicate with aggregate selections applied. In essence, instead of deriving `path` tuples for every newly derived `path`, one only needs to derive a new `path` in the recursive case whenever the `shortestPath` between any two nodes are updated. This is achieved by applying at runtime a modified evaluation strategy, that uses the selection `s1` (expressed as the last rule) as a filter for removing derived `path` tuples (grouped by `[X,Y]`) that do not change the current minimum `(C)` for each group. Note that aggregate selections optimization not only reduces the number of derivations, but it also has the nice effect of also ensuring termination, since path cycles are avoided by pruning away paths that do not contribute to the shortest paths.

3

Recursive Query Processing

Recursive query processing methods can be broadly categorized as either *bottom-up* methods, or *top-down* methods. Bottom-up methods answer a query by applying all rules of a program to ground tuples, deriving tuples that satisfy rule bodies into predicates in rule heads. The minimal model for the given program and ground tuples is explicitly materialized as a new database instance; the answer to the query is then obtained through a simple select/project/join operations over the materialized database instance. In contrast, top-down methods answer a query by pushing selection criteria (i.e. constants) from the query *down* into rules that may answer the query (i.e. rules deriving into predicates being queried), creating more (sub)queries from the atoms of these rules' bodies; the subqueries are in turn answered in a similar, top-down fashion. In this chapter, we discuss one representative method from each category: the bottom-up method of semi-naïve, and the top-down method of query/subquery (QSQ).

We illustrate the differences between these methods using the following running example, which is similar to the earlier reachability example presented in Section 1.3. Here, the EDB predicate `link` is populated with tuples `link(a,b)`, `link(b,c)`, `link(c,c)`, `link(c,d)`:

```
r1  reachable(X,Y)  :-  link(X,Y).
r2  reachable(X,Y)  :-  reachable(X,Z), link(Z,Y).
query(Y) :- reachable(b,Y).
```

As before, the distinguished query predicate (output of interest) is the `reachable` table. However, unlike the earlier example, the output is bounded to the constant `b`, indicating that we are only interested in `reachable` tuples for node `b`. As we will see later in this paper, restricting the output set has implications on the relative overheads of using bottom-up vs top-down evaluation techniques.

3.1 Bottom-up Evaluation

The least fixpoint semantics of Datalog (2.2) gives rise to a simple bottom-up evaluation algorithm: one can perform the evaluation in *iterations* starting from a base data instance containing only EDBs; in each iteration, all rules are evaluated, deriving tuples satisfying rule bodies (through immediate consequence operator T_P); the iterative evaluation stops when no new tuples can be derived. This method of evaluation is referred to as the *naïve method*. Example 2.4 illustrates the result of applying the naïve method on our running example, where the evaluation terminates in four iterations[1].

Figure 1.2 also demonstrates that many tuples are derived more than once using the naïve method: e.g. `{(a,b), (b,c), (c,c), (c,d)}` are derived four times, `{(a,c), (b,d)}` twice, etc. Such redundant derivations arise from the use of the full content of the predicates in each iteration, regardless of whether the content would generate additional tuples (with respect to the previous iteration). The *semi-naïve* method [21] aims to minimize the number of redundant derivations.

The intuition behind the semi-naïve method is that in each iteration, one should avoid repeat the computation that has already been done in previous iterations. Indeed, only the new tuples derived in the previous iteration can lead to the derivation of more tuples. Thus, the

[1]It is easy to see that the content of `reachable` in each iteration corresponds exactly to the content of `reachable` in each application of the immediate consequence operator described in 2.4.

evaluation in each iteration should focus on the newly derived tuples (called *deltas*) from the previous iteration. Additionally, it should compute in an efficient manner these *deltas* for the current iteration, for the use in the next iteration. The content of a predicate for any iteration, is simply the union of its content from the previous iteration, and the deltas from the current iteration.

Next, we show steps to systematically derive rules that use and compute deltas, and outline the semi-naïve algorithm. We apply semi-naïve to the reachability example to illustrate concretely the reduction in redundant derivations. We close the section by discussing an optimization of semi-naïve, that applies to *linearly recursive* programs.

3.1.1 The Semi-Naïve Method

We assume the following general form for a Datalog rule, where p is mutually recursive with IDB predicates p_1 through p_n, and q_1 through q_m are EDB's and built-in comparison predicates:

$$p \texttt{ :- } p_1, \ldots, p_n, q_1, \ldots, q_m\,.$$

We use $p^{[i]}$ (and $p_j^{[i]}$) to denote the set of tuples in p (respectively, p_j) at the beginning of the i^{th} iteration, starting from 0. We use $\delta(p)^{[i]}$ to denote the *new* tuples generated in iteration i (that is, for every i, $p^{[i+1]} = p^{[i]} \cup \delta(p)^{[i]}$). $\delta(p)^{[i]}$ is used as input to iteration $i+1$. For example, $p^{[0]}$ is the empty set; and $\delta(p)^{[0]}$ is those tuples derived by rules that only use EDB predicates in their bodies.

Applying the above definitions, it is easy to see that the following rule computes $p^{[i]}$ for all $i > 0$:

$$p^{[i+1]} \texttt{ :- } p_1^{[i]}, \ldots, p_n^{[i]}, q_1, \ldots, q_m\,.$$

By applying $p^{[i+1]} = p^{[i]} \cup \delta(p)^{[i]})$ to $p_1^{[i]}$ to $p_n^{[i]}$, we have

$$p^{[i+1]} \texttt{ :- } (p_1^{[i-1]} \cup \delta(p_1)^{[i-1]}), \ldots, (p_n^{[i-1]} \cup \delta(p_n)^{[i-1]}), q_1, \ldots, q_m\,.$$

Distributing $\cup$ over the conjunction, the following holds for $p^{[i+1]}$ in Figure 3.1. Here, $\Delta(p)^{[i]}$ is an over-approximation of $\delta(p)^{[i]}$: $\delta(p)^{[i]} := \Delta(p)^{[i]} - p^{[i]}$. In order to avoid redundant tuples, $\delta(p)^{[i]}$ is used in place of the true delta $\Delta(p)^{[i]}$ as input to the next iteration. Figure 3.2 presents the pseudocode for semi-naïve algorithm. Each repeat-until loop represents one iteration, where *evaluate* involves running the delta rules in Figure 3.1 at each iteration. Interested readers should refer to [21] for proofs of correctness.

$$\begin{array}{lll}
p^{[i+1]} & = & p^{[i]} \cup \Delta(p)^{[i]} \\
\Delta(p)^{[i]} & \texttt{:-} & \delta(p_1)^{[i-1]}, p_2^{[i-1]}, \ldots, p_n^{[i-1]}, q_1, \ldots, q_m\,. \\
\Delta(p)^{[i]} & \texttt{:-} & p_1^{[i]}, \delta(p_2)^{[i-1]}, p_3^{[i-1]} \ldots, p_n^{[i-1]}, q_1, \ldots, q_m\,. \\
\ldots & & \\
\Delta(p)^{[i]} & \texttt{:-} & p_1^{[i]}, \ldots, p_{n-1}^{[i]}, \delta(p_n)^{[i-1]}, q_1, \ldots, q_m\,.
\end{array}$$

Figure 3.1: Delta rules for semi-naïve evaluation

Figure 3.3 shows the tuples of `reachable` being computed using the semi-naïve algorithm, as well as the deltas. Note that the final set of `reachable` tuples are exactly the same as that in Figure 1.2. However, the semi-naïve method reduces the number of tuples derived compared to the naïve method. For instance, `reachable(b,c)` is derived twice (in $\delta(\texttt{reachable})^{[0]}$ and $\Delta(\texttt{reachable})^{[1]}$). However, it is used exactly once ($\delta(\texttt{reachable})^{[1]}$) for triggering the delta rules in iteration 1.

Rules with negation. When the semi-naïve algorithm is applied to Datalog programs with stratified negation, a predicate would only depend on a negated predicate that is computed in a lower strata. We can thus treat negated predicates as EDB's.

3.1.2 Semi-Naïve for Linearly Recursive Datalog

It is worth noting that the recursive rule in our example is *linear*: `reachable` appears exactly once in the body of the recursive rule *r2*. For linearly recursive rules, the computation of $\Delta(p)^{[i+1]}$ (and consequently, $\delta(p)^{[i]}$) can be simplified.

Let the following be the general form of a linearly recursive rule, where p is only mutually recursive with one predicate (p_j) that appears in the body of this rule:

$$p \texttt{ :- } p_1, \ldots, p_j, \ldots, p_n\,.$$

Algorithm 3.1.1: Semi-Naïve( )

```
for each IDB predicate $p$
  do { $p^{[0]} := \emptyset$
     { $\delta(p)^{[0]} :=$ tuples produced by rules using only EDB's
$i := 1$
repeat
  $p^{[i]} := p^{[i-1]} \cup \delta(p)^{[i-1]}$
  evaluate $\Delta(p)^{[i]}$
  $\delta(p)^{[i]} := \Delta(p)^{[i]} - p^{[i]}$
  $i := i + 1$
until $\delta(p)^{[i]} = \emptyset$ for each IDB predicate $p$
```

Figure 3.2: Pseudocode for semi-naïve evaluation

Applying the same definitions and rewrites as shown in Section 3.1.1, we can show that the semi-naïve algorithm can be used with the following definitions of $\Delta(p)^{[i]}$ and $\delta(p)^{[i]}$:

$$\Delta(p)^{[i]} \quad \texttt{:-} \quad p_1^{[i-1]}, \ldots, \delta(p_j)^{[i-1]}, \ldots, p_{n-1}^{[i-1]}, q_1, \ldots, q_m\,.$$
$$\delta(p)^{[i+1]} \quad \texttt{:-} \quad \delta(p_j)^{[i]}, q_1, \ldots, q_m\,.$$

3.2 Top-down Evaluation

While semi-naïve minimizes the redundant derivation of the same tuples across multiple iterations, it does not minimize the derivation of tuples that are not necessary in answering a query. For instance, the tuple `reachable(a,b)` does not participate in answering the query `reachable(b,Y)` in any way. Top-down methods aim to derive only those tuples relevant to the query, by starting the evaluation from the query itself, and pushing selection criteria (i.e. constants) from the query into rules. One can think of top-down evaluation as the search for proof trees for the queries, per the proof-theoretic semantics of Datalog described in Section 2.2. The tuples derived by top-down methods are exactly those that appear in the proof tree.

In this section, we discuss the representative top-down method, *Initialization*

$\texttt{reachable}^{[0]} := \emptyset$

$\delta(\texttt{reachable})^{[0]} := \{(\texttt{a},\texttt{b}),(\texttt{b},\texttt{c}),(\texttt{c},\texttt{d}),(\texttt{c},\texttt{c})\}$

Iteration 1:

$\texttt{reachable}^{[1]} := \{(\texttt{a},\texttt{b}),(\texttt{b},\texttt{c}),(\texttt{c},\texttt{d}),(\texttt{c},\texttt{c})\}$

$\Delta(\texttt{reachable})^{[1]} := \{(\texttt{a},\texttt{c}),(\texttt{b},\texttt{d}),(\texttt{b},\texttt{c}),(\texttt{c},\texttt{c})\}$

$\delta(\texttt{reachable})^{[1]} := \{(\texttt{a},\texttt{c}),(\texttt{b},\texttt{d})\}$

Iteration 2:

$\texttt{reachable}^{[2]} := \{(\texttt{a},\texttt{b}),(\texttt{b},\texttt{c}),(\texttt{c},\texttt{d}),(\texttt{c},\texttt{c}),(\texttt{a},\texttt{c}),(\texttt{b},\texttt{d})\}$

$\Delta(\texttt{reachable})^{[2]} := \{(\texttt{a},\texttt{d})\}$

$\delta(\texttt{reachable})^{[2]} := \{(\texttt{a},\texttt{d})\}$

Iteration 3:

$\texttt{reachable}^{[3]} := \{(\texttt{a},\texttt{b}),(\texttt{b},\texttt{c}),(\texttt{c},\texttt{d}),(\texttt{c},\texttt{c}),(\texttt{a},\texttt{c}),(\texttt{b},\texttt{d}),(\texttt{a},\texttt{d})\}$

$\Delta(\texttt{reachable})^{[3]} := \emptyset$

$\delta(\texttt{reachable})^{[3]} := \emptyset$

Figure 3.3: The computation of `reachable` using the semi-naïve algorithm

Query/Subquery (QSQ)[67]. We first provide an intuition for QSQ by showing how it answers the query in our running example. We then describe key components of a (sub)query, and outline the iterative QSQ algorithm[2].

3.2.1 Query-Subquery By Example

QSQ evaluation of a query begins by *unifying* the distinguished query atoms with head atoms of rules. We refer to rules whose head atoms unify with the distinguished query atoms as *candidate rules*. In our running example (shown in Section 1.3), both *r1* and *r2* are candidate rules. Unification introduces constant bindings for variables in the candidate rules. For instance, unifying `reachable(b,Y)` with the head of *r1*, `reachable(X,Y)`, introduces the binding `b` for variable `X`. We denote this binding as $\{\texttt{X} \mapsto \texttt{b}\}$. We say that the first argument of `reachable`, in this context, is *bound*. We refer to the passing of binding information

[2]Readers can find the description of recursive QSQ in Chapter 13 of [12]

through unification as *top-down information passing*.

Next, binding information is pushed into the bodies of candidate rules, such that subqueries can be constructed from atoms in the bodies. For instance, pushing $\{\texttt{X} \mapsto \texttt{b}\}$ into the body of rule *r2*, results in the subquery `link(b,Z)`. The evaluation of this subquery consequently introduces more binding information, $\{\texttt{Z} \mapsto \texttt{c}\}$. When creating subqueries from the subsequent body atom, `reachable(Z,Y)`, binding information from both the head atom, as well as the evaluation of the subquery `link(b,Z)`, is used. The resulting subquery is thus `reachable(c,Y)`. We refer to the passing of binding information from atom to atom in the same rule body as *sideways information passing*[3].

Subqueries are answered similarly, by unifying them with candidate rules, possibly resulting in more subqueries. When all subqueries pertaining to a rule body are answered, QSQ produces an answer set for the (sub)query pertaining to the rule head. For instance, the subquery in *r1*'s body, `link(b,Y)` can be answered immediately by looking up the tuples in EDB `link`. This produces the tuple {`(b,c)`}, which is placed in the answer set for query `reachable(b,Y)`.

The production and answering of queries/subqueries repeat until no more tuples are derived into answer sets, and no more subqueries are produced.

3.2.2 Query-Subquery Evaluation

There are two key components in the construction of a (sub)query:

- A predicate where certain arguments are *bound*. The boundedness of an argument indicates whether it should receive binding informations—i.e., be replaced by constants—in the construction of subqueries. For every predicate P, we use an *adorned predicate*, P^{γ}, to represent P with certain arguments bound.
- The binding information passed down, or sideways, into the bounded arguments. We use R to represent *binding relations*.

[3]Sideways information passing is exploited in an optimization strategy for bottom-up evaluation, called *magic sets*. We discuss magic sets in detail in Section 3.3.

Essentially, R is a set of all possible bindings for the bounded arguments in P^γ. For instance, $\{\{\texttt{X} \mapsto \texttt{b}, \texttt{Z} \mapsto \texttt{c}\}, \{\texttt{X} \mapsto \texttt{b}, \texttt{Z} \mapsto \texttt{a}\}\}$ represents two possible bindings for `X` and `Z`.

Together, $\langle P^\gamma, R\rangle$ represent a set of subqueries to be answered. These subqueries are constructed by replacing the bounded arguments of P^γ with constants, according to the binding information in R. Next, we discuss how adorned predicates and binding relations are computed; we then show their use in QSQ evaluations.

Adorned predicates. Given a predicate P, we refer to P^γ as the *adorned* version of P [117]. γ is a sequence of b's and f's, where the length of γ is exactly the arity of P. Thus, each argument of P has a corresponding b or f in γ. A b denotes that the corresponding argument of P is bound—i.e. it is expecting binding information—and f denotes otherwise. For example, $\texttt{reachable}^{bf}$ is an adorned version of `reachable`, and expects bindings for its first argument.

Furthermore, we rewrite rules deriving P into *adorned rules* that derive P^γ. The rewritten rules are used to determined how subqueries should be constructed, in order to answer queries $\langle P^\gamma, R\rangle$.

To get an adorned rule, every atom in the original rule body is rewritten to refer to an adorned predicate. The adornment is determined as follows: an argument position is bound if it is a constant, a variable bound in the rule head as indicated by the adornment of the head atom, or a variable bound by some atom to its left in the rule body. Intuitively, this corresponds to passing the binding information from the rule head to the rule body. For instance, the adorned rules deriving $\texttt{reachable}^{bf}$ are as follows:

```
ar1  reachable^bf(X,Y) :- link^bf(X,Y).
ar2  reachable^bf(X,Y) :- link^bf(X,Z), reachable^bf(Z,Y).
```

Binding relations. There are two types of binding information in QSQ: bindings passed top-down from queries to the head of rules through unification, and bindings passed sideways from atom to atom in the body of the same rule.

We use *input relations* to represent binding information passed top-down. The input relation $input\_P^{\gamma}(V)$ denotes binding relations passed down into the head of adorned rules deriving P^{γ}. V is a set of bounded arguments in P^{γ}. Intuitively, $\langle P^{\gamma}, input\_P^{\gamma}\rangle$ represent subqueries created through unification. For instance, *input*_`reachable`^{bf}`(X)` is the input relation for rules *ar1* and *ar2*, and can be used to create subqueries from `reachable`^{bf}`(X,Y)`, by replacing `X` with constants.

We use *supplementary relations* to represent binding information passed sideways. The relation $sup^{i}_{j}(V)$ denotes binding information passed sideways into the jth atom (starting from 0) in the body of rule ari, where V is a set of arguments that are (1) already bounded (by the atoms before j), and (2) later referenced (by the head atom or body atoms on or after j). Intuitively, $sup^{i}_{j}(V)$ contains binding information used to construct subqueries from the j^{th} atom. For instance, for *ar2*, we have $sup^{2}_{0}(X)$, which provides the constant bindings for `X` that can be used to construct subqueries from `link`^{bf}`(X,Z)`, and $sup^{2}_{1}(X, Z)$ for constructing subqueries from `reachable`^{bf}`(Z,Y)` (since `Z` is now bounded by `link`^{bf}`(X,Z)`).

For a rule containing n atoms in the body, the supplementary relation $sup^{i}_{n}(V)$ denotes the resulting binding information from evaluating all subqueries constructed from the rule body. Thus, $sup^{i}_{n}(V)$ contains the answer set for the head query. For example, for rule $ar2$ in our example, relation $sup^{2}_{2}(X, Y)$ contains the answers to `reachable`^{bf}`(X,Y)`.

For every adorned predicate P^{γ}, let ans_P^{γ} be answers to queries constructed using P^{γ}. ans_P^{γ} is thus populated using $sup^{i}_{n}(V)$, for all rules i deriving into P^{γ}.

3.2.3 Steps of QSQ Evaluation

There are four steps of QSQ evaluation:

Step 1. The evaluation begins by unifying the query atom with adorned rules. The binding of arguments in the query atom dictates which adorned rules the atom can unify with. The input relation for the unified predicate is populated with bindings produced by a successful unification.

Step 2. The 0th supplementary relation for a candidate rule is computed by projecting out the necessary variables from the input relation.

Step 3. The production and evaluation of subqueries, and the computation of subsequent supplementary relations, are interleaved.

(a) Let the j^{th} atom in rule i's body be $P^{\gamma}(V)$. We construct subqueries from $\langle P^{\gamma}, sup^i_j(V')\rangle$, By substituting variables in $P^{\gamma}(V)$ with constants from $sup^i_j(V')$.

(b) A subsequent supplementary relation $sup^i_{j+1}(V)$ is computed by joining the answer set of the j^{th} atom, with $sup^i_j(V')$.

(c) As subqueries are constructed from $\langle P^{\gamma}, sup^i_j(V')\rangle$, it is necessary to pass the binding information from $sup^i_j(V's)$ to $input\_P^{\gamma}$, such that these subqueries can be evaluated using these same evaluation steps (notably, step (1) requires that $input\_P^{\gamma}$ contains the appropriate bindings). Thus, we compute $input\_P^{\gamma}$ by projecting out the appropriate variables from $sup^i_j(V's)$.

Step 4. The final supplementary relation in a rule is used to compute the answers for the rule head.

Example 3.1. We next illustrate the steps of QSQ by applying them to answer `query reachable(b,Y)`.

Step 1: First, we obtain the input relation, $input\_\texttt{reachable}^{bf}\texttt{(X)}$, by unifying the query with the head of both candidate rules. This results in an input relation that contains the binding $\{\{\texttt{X} \mapsto \texttt{b}\}\}$.

Step 2: Next, we obtain the supplementary relations $sup^1_0(\texttt{X})$ and $sup^2_0(\texttt{X})$ by projecting out the necessary (and only) variable from the input relation. In this case, both supplementary relations contain a single binding relation $\{\{\texttt{X} \mapsto \texttt{b}\}\}$.

Step 3. There supplementary relations allow us to obtain subgoals and subsequent supplementary relations.

(a) For rule *ar1*, we substitute the bound variable in $\texttt{link}^{bf}\texttt{(X,Y)}$ with $sup^1_0(\texttt{X})$, and obtain the subgoal `link(b,Y)`. The answer set for `link(b,Y)` of rule *ar1* is $\{\{\texttt{X} \mapsto \texttt{b}, \texttt{Y} \mapsto \texttt{c}\}\}$, which directly goes into the answer set for *ans*_`reachable(X,Y)` in step 4. For rule *ar2*, we similarly obtain the subgoal `link(b,Z)`. The answer set for `link(b,Z)` is used to produce the next supplementary relation, sup_1^2(`Z`), which contains $\{\{$`Z` $\mapsto$ `c`$\}\}$.

(b) We can now use sup_1^2(`Z`), and `reachable`^{bf}`(Z,Y)`, to produce the next subgoal: `reachable(c,Y)`.

(c) Since we used sup_1^2(`Z`) to create a new subgoal, we need to similarly enhance *input*_`reachable`^{bf}`(X)` with the same binding information (Step 3c). After this step, *input*_`reachable`^{bf}`(X)` contains $\{\{$`X` $\mapsto$ `b`$\}, \{$`X` $\mapsto$ `c`$\}\}$. The subquery `reachable(c,Y)` can then be evaluated using the same steps going back to Step 2. The result would be used to enhance *ans*_`reachable(X,Y)` in step 4.

Step 4. Use *ans*_`reachable(X,Y)` generated in Step 3 as answers for the rule head.

The QSQ method can be implemented iteratively, by simply iterating over the above 4 steps until no more subqueries are constructed, and the answer set for all predicates no longer change. An alternative method is recursive QSQ. Interested readers may consult [23] for an exposition of these algorithms.

3.3 Magic Sets

Top-down evaluation methods (in particular QSQ) that derive only facts relevant for answering queries, by starting the evaluations from the queries, pushing binding information through the evaluation of rules. There has been a lot of research on allowing bottom-up evaluation methods to take advantage of the information in queries. *Magic sets* [25] is a family of rewrite techniques that, given a Datalog program P, produces another Datalog program P' where, for any instance I and query q, the answer for q computed by P' is exactly that computed by P. Furthermore, the semi-naïve method, when applied to P', derives exactly the same set of facts derived by QSQ. The insight behind a magic sets rewrite is that binding information from queries can be represented as predicates; these predicates can be inserted into rule bodies, forcing joins that will constrain the facts derived in a bottom-up evaluation of the rule. There is a strong connection between magic sets techniques and QSQ: the same binding information represented by input and supplementary relations in QSQ, are precisely the information needed to constrain bottom-up evaluations in magic sets. The key is to derive these relations using (bottom-up evaluated) Datalog rules, and use them to pass binding information *sideways* into rule bodies, removing the need for top-down information passing.

3.3.1 Basics of a Magic Sets Rewrite

There are three steps common to all magic sets rewrites:

(1) Rewrite rules into their adorned versions, where the adornments needed are determined by queries.

(2) Define binding predicates and rules that derive into them.

(3) Rewrite adorned rules by adding binding predicates in rule bodies.

We next describe these three steps informally, focusing on intuition rather than formal notation. We refer interested readers to [25] for a more formal treatment. We illustrate the application of magic sets to the same example we used to illustrate QSQ (Section 3.2.2), to show the strong correspondence between the two methods:

```
r1 reachable(X,Y) :- link(X,Y).
r2 reachable(X,Y) :- link(X,Z), reachable(Z,Y).
query(Y) :- reachable(b,Y).
```

Adorned rules. The rewrite to produce adorned rules follows the same rules as those described in Section 3.2.2. Note that a rule deriving into a predicate of arity n has 2^n adorned versions. It is not necessary to produce all adorned rules; they can be produced by need based on the queries being answered. In our example, since the query only calls for `reachable`^{bf}, we produce the following adorned rules:

```
ar1 reachable^bf(X,Y) :- link^bf(X,Y).
ar2 reachable^bf(X,Y) :- link^bf(X,Z), reachable^bf(Z,Y).
```

Deriving binding predicates. Recall that two types of relations are used to constrain the top-down evaluation of this program: the input relation *input_*`reachable`^{bf}`(X)`, and the supplementary relations $sup^i_j(V)$ for every atom j in rule i. A magic sets rewrite defines these supplementary relations as predicates derived using Datalog rules.

First, the input relation is initialized with constants from queries:

m1 *input_*`reachable`^{bf}`(b).`

Next, we define rules for supplementary relations. As we described in Section 3.2.2: the $0th$ supplementary relation is computed by projecting out the appropriate variables from the input relation; a subsequent supplementary relation is computed by joining the supplementary relation and atom to its left in the rule body. For our example, the following rules derive the supplementary relations:

m2 sup^1_0`(X) :- `*input_*`reachable`^{bf}`(X).`
m3 sup^2_0`(X) :- `*input_*`reachable`^{bf}`(X).`
m4 sup^2_1`(X,Z) :- `sup^2_0`(X), link(X,Z).`

Rewrite rules using binding predicates. We now rewrite adorned rules, such that the j^{th} atom in the body of rule i is preceded by the supplementary predicate $sup^i_j(V)$. The intuition is that the tuples in $sup^i_j(V)$ constrain the evaluation of the rule by joining with the j^{th} atom, producing exactly those facts needed to derive the necessary facts into the rule head. The following are the rewritten rules for `reachable`^{bf}:

m5 `reachable`^{bf}`(X,Y) :- `sup^1_0`(X), link(X,Y).`
m6 `reachable`^{bf}`(X,Y) :- `sup^2_0`(X), link(X,Z), `sup^2_1`(X,Z), reachable`^{bf}`(Z,Y).`

Of course, the evaluation of the predicate for the j^{th} atom itself needs to be properly constrained. This is done by populating its input predicate, using $sup^i_j(V)$ (Note the correspondence between the rule below to step 3c of QSQ):

m7 *input_*`reachable`^{bf}`(X) :- `sup^2_1`(X,Z).`

Together, rules *m1* through *m7* constitute a rewritten version of our original program. Applying the semi-naïve algorithm to this rewritten program, it is easy to see that only facts relevant to the query, i.e. `reachable(b,Y)`, are derived.

Note that the input and supplementary predicates are also referred to as *magic predicates* in the literature. We were inspired by [12] to keep the presentation of magic sets and QSQ similar, to emphasize their close connection.

3.3.2 Effective Applications of Magic Sets

In applying a magic sets rewrite to a Datalog program, there can be many different strategies in the definition of binding predicates and their use in adorned rules. For instance, in the example shown in the previous section, since sup_1^2(X,Z) is computed by joining sup_0^2(X) and `link(X,Z)`, it is an equally correct rewrite to replace *m6* with the following rule:

m6' reachable^{bf}(X,Y) :- sup_1^2(X,Z), reachable^{bf}(Z,Y).

Yet another strategy is to reorder the atoms in the original rule bodies, before even applying the rewrite steps. For instance, the following version of *r2* would have yielded a rather different rewrite result:

```
reachable(X,Y) :- reachable(Z,Y), link(X,Z).
```

These strategies are referred to as *sideways information passing strategies*, or SIPS. A poorly chosen SIPS not only does not improve the evaluation performance of a program; it can degrade it through the additional cost of computing binding predicates and joining them in rule bodies.

To see concretely the effect of a poorly chosen SIPS, consider the example program in Figure 3.4, drawn from a program analysis application [108]. The program queries for all subtypes of `Cloneable` that do not declare a method `clone`. Transformed as is, no binding information is available for `hasSubtypePlus`. Yet the rules for `hasSubtypePlus` are the most costly, as they compute the transitive closure of the `hasSubtype` relation. Ideally, we would like to only compute what is

```
r1  query(C) :- hasSubtypePlus(Cloneable,C),
       not declaresMethod(C, "clone"),
       type(Cloneable), hasName(Cloneable, N), N = "Cloneable".
r2  hasSubtypePlus(SUPER,SUB) :- hasSubtype(SUPER,SUB).
r3  hasSubtypePlus(SUPER,SUB) :- hasSubtypePlus(SUPER,MID),
       hasSubtype(MID,SUB).
r4  hasName(C,N) :- className(C, N), interfaceName(C, N).
r5  declaresMethod(C,Name) :- method(M,C), methodName(M,Name).
```

Figure 3.4: Program to be optimized using magic sets

necessary for this query, i.e. those subtypes of `Cloneable`. Furthermore, the computation of `declaresMethod` is not meaningfully constrained, either. Its binding predicate would contain the entire first column of `hasSubtypePlus`. Indeed, the cost of evaluating the magic sets rewritten version of this program as is would have no benefit over evaluating the original program. It would most likely cause performance degradation, as the rules for binding predicates would also need to be evaluated.

The key in choosing an effective SIPS is to maximize the selectivity of the binding predicates, while minimizing the cost of their computation. In the remainder of this section, we discuss three factors in effectively choosing a SIPS, and show concretely how they help in achieving a performance-enhancing rewrite of the program in Figure 3.4:

- Determining the order of atoms in queries and rules, such that expensive computations can be effectively constrained.
- Choosing the atoms used to compute binding predicates, such that their computation cost is justified by their selectiveness.
- Choosing which variables are considered bound in each atom (i.e. the variables stored in binding predicates), to minimize the size of (and thus the cost of materializing) binding predicates.

Determining the order of atoms. There is a strong relationship between choosing an atom ordering as part of a SIPS, and choosing a join ordering for query evaluation: both attempt to minimize the cost of evaluation by passing selective binding information through variables into the subsequent atoms. This connection has been explored by several magic sets implementations (e.g., [109, 89, 108]). We discuss two representative approaches: one makes use of runtime information; the other is a static approach, independent of database runtime.

Runtime Approach. Seshadri et.al.[109] proposes a runtime approach in which the ordering of atoms is optimized for the purpose of computing binding predicates for magic sets rewrites. The key idea behind this approach is that, computing a binding predicate, and using it in the body of a query, can be modeled as a relational operator. The query optimizer can choose to incorporate this operator into its plan if it lowers the overall cost of query evaluation.

The relational operator "filter-join" is introduced for this purpose. A filter-join of two relations, R filter-join S, means that the join attributes of R are projected out into a "filter" relation; only those tuples of S that join with the filter relation are then joined with R. Thus, a filter-join represents the creation of a binding predicate, and adding the binding predicate to a query/rule body to constrain its evaluation.

If the query plan determined by the query optimizer contains a filter-join, that means creation and use of a binding predicate lowers the overall cost of computation—that its selectivity outweighs the cost of its computation. The query plan specifies exactly which atoms to use to create supplementary relation, and in what order.

Note that this approach has been implemented only for non-recursive queries, as part of the IBM DB2 database.

Static Approach. Sereni et.al. [108] proposes a purely static approach, and requires no modifications to a database runtime. This approach relies on the approximate statistics on the contents of predicates—even those of IDB predicates, computed using abstract interpretation [38]. A heuristic, similar to ones implemented by runtime query optimizers, then uses this approximated information to find an ordering that minimizes cost.

Central to this approach is the notion of a *predicate dependency graph*: a static representation of statistics on a predicate. A predicate dependency graph for a predicate $P(X_1, ..., X_n)$ is a triple (Σ, G, Π):

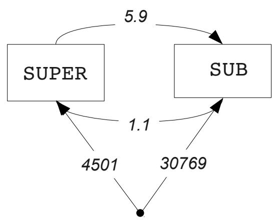


Figure 3.5: Dependency of `hasSubtype`

- Σ is a function that assigns an estimated size for each X_i.
- G is a set of arrows $X_i \xrightarrow{\alpha} X_j$, denoting that for every value of X_i, there are α values of X_j.
- Π is a set of equality constraints of the form $X_i = X_j$.

Figure 3.5 illustrates the dependency graph for the predicate `hasSubtype(SUPER,SUB)`. The labeled edges between columns `SUPER` and `SUB` denote that for every superclass (`SUPER`), there are 5.9 subclasses (`SUB`), and every sub-class has on average 1.1 super-classes. The labeled edges between the root node and the columns denote that `SUPER` has 4501 unique values (Σ(`SUPER`)), and `SUB` 30769 unique values.

Whereas dependency graphs for EDB predicates are derived by analyzing ground facts stored in them, dependency graphs for IDB predicates need to be derived from EDB dependency graphs. [108] defines a method based on *abstract interpretation* [38] for computing IDB dependency graphs.

An *abstract interpreter* for Datalog interprets an IDB rule using *abstract values* (that is, pessimistic upper bounds) for predicates in the rule body, to derive abstract values for the predicate in the rule head. This is analogous to a Datalog runtime that interprets an IDB rule using concrete values of predicates in the rule body, to derive concrete values for the predicate in the rule head. In this application, the abstract values are predicate dependency graphs.

To provide some intuition for this technique, we next show the abstract interpretation of an intersection, $\phi_1 \wedge \phi_2$. We refer interested readers to [108] for the interpretation of other relational operators.

Let $[\![\phi_i]\!] = (\Sigma_i, G_i, \Pi_i)$ denote the dependency graph for formula ϕ_i. $[\![\phi_1 \wedge \phi_2]\!] = (\Sigma, G, \Pi)$ is determined as follows:

- $\Sigma(X)$ for any column X in $\phi_1 \wedge \phi_2$ is no bigger than either $\Sigma_1(X)$ or $\Sigma_2(X)$. Thus, $\Sigma(X) = min(\Sigma_1(X), \Sigma_2(X))$.
- In the case that G_1 and G_2 have different constraints on the same column, e.g., $X \overset{\alpha_1}{\to} Y \in G_1$, $X \overset{\alpha_2}{\to} Y \in G_2$, we note that both α_1 and α_2 are pessimistic approximations. Thus, we can pick the best, or the smaller, of the two as the result of the interpretation: $X \overset{\alpha}{\to} Y \in G$, where $\alpha = min(\alpha_1, \alpha_2)$.
- All equality constraints hold after intersection: $\Pi = \Pi_1 \cup \Pi_2$.

In the presence of recursion, [108] unrolls a recursive cycle some fixed number of times (3 proved effective in practice), and interprets the resulting, non-recursive program.

Given predicate dependency graphs for all predicates, we can compute the estimated size of any given predicate: it is the weight of the minimum spanning tree of the predicate's dependency graph, where edge weights are composed with multiplication rather than addition. Note that for `hasSubtype` (Figure 3.5), there are two reasonable estimated sizes: $4501 \times 5.9 \approx 26556$, or $30769 \times 1.1 \approx 33846$. This discrepancy arises from the estimated nature of this analysis. Since the analysis stems from pessimistic estimates, both numbers are likely to be over-estimates. Therefore, one might choose the lesser of the two as the estimated size of `hasSubtype`.

Using the estimated predicate size information, a greedy heuristic is used to order the atoms in the query, by picking the smallest predicate first. The size of a predicate is computed *in context*. For instance, in reordering the query in Figure 3.4, we pick `N = "Cloneable"` first, as it is guaranteed to have size 1. Next, we pick `hasName(Cloneable,N)`. Even though the estimated size of the entire `hasName` predicate may be much larger, in context of the atoms before it, its size is the number of types with the name `"Cloneable"`, a much smaller number. Informally, the estimated size of formula ϕ in the context of ψ, is simply the estimated size of the conjunction $\phi \wedge \psi$.

Applying the static technique, we can reorder query as follows. We observe that this ordering would provide exactly the binding necessary to constrain the expensive computations of `hasSubtypePlus`:

```
r1  query(C) :- N = "Cloneable", hasName(Cloneable, N),
      type(Cloneable), hasSubtypePlus(Cloneable,C),
      not declaresMethod(C, "clone").
```

Note that the general technique of abstract interpretation can be applied over other abstract domains to provide potentially better static estimates of predicate statistics. For instance, [71] uses abstract interpretation over a domain of histograms.

Defining binding predicates. A binding predicate for an atom can be created using all, or some subset of atoms appearing to its left in the rule body. To minimize the cost of computing the binding predicate, one should only include those atoms that make the binding predicate more selective. For instance, the binding predicate for `hasSubtypePlus` in *r1* can be defined as follows:

sup_3^1`(Cloneable) :- N = "Cloneable",`
```
      hasName(Cloneable, N), type(Cloneable).
```

However, `type(Cloneable)` does not increase the selectiveness of the binding predicate: `hasName(Cloneable,N)` implies that `Cloneable` is either a `class` or `interface`, both of which are `type`'s. Thus, the following binding predicate is equally selective, but less costly:

sup_3^1`( Cloneable) :- N = "Cloneable", hasName(Cloneable, N).`

The query plan produced by [109] specifies which atoms to include in a binding predicate: i.e. R filter-join S means that the atoms used to produce R should be used to create the binding predicate.

The static approach of [108] applies the same technique used for atom ordering, to determining which atoms to include in the production of a binding predicate. If the estimated size of $\phi \wedge \psi$ is the same (or within a certain threshold of) as that of ϕ, then ψ is not adding more selectivity, and thus not necessary in the production of a binding predicate.

Choice of bound variables. Bound variables are those variables projected into binding predicates. Naively, bound variables are all those that appear in the atoms used to derive a binding predicate. However, this is an overly conservative strategy, as some variables may not constrain the computation at all. Consider the following program:

```
p(X,Y) :- X = 1, type(Y).
q(X,Y) :- X = Y.
query(X,Y) :- p(X,Y), q(X,Y).
```

The fact that `p` is constraining `q` arises only from the binding of `X` to a constant. `Y` is not constrained at all, and thus does not need to be projected into the binding predicate for `q`.

3.3.3 Extensions to Magic Sets

Magic sets techniques have been extended to pass other useful constraining information through rewriting. Notably, there has been a line of work applying magic sets-like techniques to bottom-up evaluated logic programs with arithmetic comparison constraints (e.g., $X < Y$, $X >= Z$, etc.) For instance, the technique presented in [111] allows us to optimize the following program:

```
r1  p(X,Y) :- q(X,Y).
r2  query(X,Y) :- p(X,Y), X >= 4, X + Y < 10.
```

The constraints `X >=4` and `X+Y < 10` can be pushed into *r1*, by rewriting *r1* into the following rule, and thus constraining the derivation of `p` to facts relevant to the query:

```
r1'   p(X,Y) :- q(X,Y), X >= 4, X + Y < 10.
```

Other work [114, 31] further extends constraint pushing to handle more expressive constraints, or constraints whose values are only available at runtime.

4

Incremental Maintenance

Materialized views can be used to improve the performance of Datalog programs that perform frequent yet complex queries. If the EDBs that a materialized view is computed from change, the materialized view needs to be maintained to reflect such changes. The efficient maintenance of materialized views thus become an important aspect of sustaining their performance benefits. Incremental view maintenance refers to a family of methods aimed at the efficient maintenance of views in the face of base relation changes. Rather than re-computing materialized views in full, incremental view maintenance methods only compute the changes to views based on changes to EDBs.

In this chapter, we introduce three incremental maintenance algorithms. *Counting* [52] is an effective method for incrementally maintaining non-recursive view definitions. It counts the possible ways a tuple can be derived into a view; tuples with count of at least 1 are part of the view; tuples with count of 0 are deleted from the view. The *Delete and Rederive* (DRed) [52] method is well-known for maintaining recursive views. DRed builds on semi-naïve evaluation, and works by first *over-deleting* tuples conservatively, and then *re-deriving* tuples that may have alternative derivations. We conclude with a provenance-based approach [87] that reduces the overhead of unnecessary deletion and rederivations of DRed, via some additional bookkeeping of the provenance of each derivation.

Note that the algorithms presented either incur storage overheads in book-keeping, or computational overheads in evaluating additional rules (and thus potentially expensive joins) in computing increments. There are certainly situations when full re-computation would be cheaper than incremental maintenance: for instance, if the majority of base tuples are deleted.

4.1 Counting Algorithm for Non-recursive Queries

As its name suggests, the counting algorithm works by storing the number of alternative derivations for each tuple t in the materialized view. We refer to this number as *count(t)*. Using Figure 1.1 in Chapter 1.3 as our example network, the initial base tuples are `link(a,b)`, `link(b,c)`, `link(c,d)`, and `link(c,c)`. When executing the all-pairs reachability program in Section 1.3 to a fixpoint, the `reachable` relation consists of the tuples `{ (a,b), (b,c), (c,c), (c,d), (a,c), (a,d), (b,d)}`. The `reachable` tuples `{(a,b),(b,c),(c,c)}` have a unique derivation each, e.g. *count(*`reachable(a,b)`*)*= 1. On the other hand, the tuples `{(c,d),(a,c),(b,d),(a,d)}` have two possible derivations each. For example, `reachable(b,c)` can be derived in two ways, from `link(b,c)`, or from `link(b,c), reachable(c,c)`. Thus, *count(*`reachable(b,c)`*)*= 2.

To implement the counting algorithm, one can use the semi-naïve algorithm described in the previous section, and maintain a count with each tuple. For each derivation or deletion of a tuple t, its *count(t)* is incremented/decremented by 1. A tuple with count of 0 is deleted. Suppose `link(a,b)` is deleted, this will result in the deletion of `reachable(a,b)`, since *count(*`reachable(a,b)`*)*= 1. On the other hand, if `link(c,c)` is deleted, even though `reachable(c,c)` is deleted, the tuples `reachable(a,c)` which has a count of 2 will not be deleted. Instead, the algorithm will simply decrement the count of `reachable(a,c)` to 1. At a later time, if `link(b,c)` is deleted, the count of `reachable(a,c)` goes to 0, and `reachable(a,c)` is deleted.

The counting algorithm may result in infinite counts in the presence of recursively derived predicates. For instance, if the underlying input graph has cycles in the paths, our `reachable` program will not terminate since each invocation of the recursive rule `r1` will generate a count of increasing value.

4.2 Delete and Re-Derive Algorithm (DRed)

Delete-and-Rederive (DRed) [52] is the standard algorithm for recursive view maintenance. DRed has three phases: deletion, put-back, and assertion. At a high level, a tuple is deleted during the deletion phase if at least one of the rules that derived it can no longer do so; the put-back phase computes the deleted tuples that have at least one other possible derivation, and thus should be put-back into the view; and lastly, the assertion phase computes all new tuples for the view due to assertions into the base relations. We illustrate DRed by applying it to the same network reachability example used by the previous section, by deleting the tuple `link(c,c)`. We focus on the deletion and put-back phase, as the assertion phase is a straightforward adaptation of the semi-naïve algorithm. Our presentation of DRed is example-driven, in order to provide the high-level intuition. More details are available in Gupta et.al. [52].

Deletion phase. To compute the deleted tuples from `reachable`, we apply delta rules similar to those for semi-naïve evaluation (Figure 3.1 in Section 3.1.1), with two adjustments. First, we initialize $\texttt{reachable}^{[0]}$ to the content of `reachable` before the deletion phase (rather than the empty set $\emptyset$). Secondly, as there are no assertions in this phase, we use $\delta^{-}(\texttt{reachable})^{[i]}$ to denote the difference between iteration $i+1$ and i, i.e, the deletion tuples newly derived in each iteration i. For base relations such as `link`, $\delta^{-}(\texttt{link})$ represent the tuples deleted from the relation, and is the same for all stages i. For our example, the deletion rules are as follows:
Initialization:

$$\delta^{-}(\texttt{reachable(X,Y)})^{[0]} \quad \texttt{:-} \quad \delta^{-}(\texttt{link(X,Y)}).$$

Initialization

$$\begin{array}{lcl} \texttt{reachable}^{[0]} & := & \{(\texttt{a},\texttt{b}),(\texttt{b},\texttt{c}),(\texttt{c},\texttt{d}),(\texttt{c},\texttt{c}),(\texttt{a},\texttt{c}),(\texttt{a},\texttt{d}),(\texttt{b},\texttt{d})\} \\ \delta^-(\texttt{reachable})^{[0]} & := & \{(\texttt{c},\texttt{c})\} \end{array}$$

Iteration 1:

$$\begin{array}{lcl} \texttt{reachable}^{[1]} & := & \{(\texttt{a},\texttt{b}),(\texttt{b},\texttt{c}),(\texttt{c},\texttt{d}),(\texttt{a},\texttt{c}),(\texttt{a},\texttt{d}),(\texttt{b},\texttt{d})\} \\ \delta^-(\texttt{reachable})^{[1]} & := & \{(\texttt{b},\texttt{c}),(\texttt{c},\texttt{d})\} \end{array}$$

Iteration 2:

$$\begin{array}{lcl} \texttt{reachable}^{[2]} & := & \{(\texttt{a},\texttt{b}),(\texttt{a},\texttt{c}),(\texttt{a},\texttt{d}),(\texttt{b},\texttt{d})\} \\ \delta^-(\texttt{reachable})^{[2]} & := & \{(\texttt{a},\texttt{c}),(\texttt{b},\texttt{d})\} \end{array}$$

Iteration 3:

$$\begin{array}{lcl} \texttt{reachable}^{[3]} & := & \{(\texttt{a},\texttt{b}),(\texttt{a},\texttt{d})\} \\ \delta^-(\texttt{reachable})^{[3]} & := & \{(\texttt{a},\texttt{d})\} \end{array}$$

Iteration 4:

$$\begin{array}{lcl} \texttt{reachable}^{[4]} & := & \{(\texttt{a},\texttt{b})\} \\ \delta^-(\texttt{reachable})^{[4]} & := & \emptyset \end{array}$$

Figure 4.1: Iterations of the deletion phase, for reachability example.

For each i >= 0:

$$\begin{array}{rcl} \delta^-(\texttt{reachable(X,Y)})^{[i+1]} & \texttt{:-} & \texttt{link(X,Z)},\ \delta^-(\texttt{reachable(Z,Y)})^{[i]} \\ \texttt{reachable}^{[\texttt{i+1}]} & := & \texttt{reachable}^{[\texttt{i}]} - \delta^-(\texttt{reachable})^{[\texttt{i}]} \end{array}$$

We apply these rules repeatedly until $\delta^-(\texttt{reachable})^{[i]}$ is $\emptyset$. Figure 4.1 illustrates the iterations of the deletion phase, as applied to the reachability example.

Put-back phase. The put-back phase re-derives tuples that may have been deleted conservatively during the first phase, but have alternate derivation paths. In our example, the put-back rules are as follows, where $\delta^-(\texttt{reachable})$ denotes the union of $\delta^-(\texttt{reachable})^{[i]}$ for all iterations i during the deletion phase:

Initialization:

$$\delta^+(\texttt{reachable(X,Y)})^{[1]} \quad \texttt{:-} \quad \texttt{link(X,Y)}.$$

For each i >= 0:

$$\begin{array}{rcl} \delta^+(\texttt{reachable(X,Y)})^{[i+1]} & \texttt{:-} & \delta^-(\texttt{reachable(X,Y)}),\texttt{link(X,Y)}. \\ \delta^+(\texttt{reachable(X,Y)})^{[i+1]} & \texttt{:-} & \delta^-(\texttt{reachable(X,Y)}),\texttt{link(X,Z)}, \\ & & \delta^+(\texttt{reachable(Z,Y)})^{[i]} \\ \texttt{reachable}^{[\texttt{i+1}]} & := & \texttt{reachable}^{[\texttt{i}]} \bigcup \delta^+(\texttt{reachable})^{[\texttt{i}]} \end{array}$$

Similar to the deletion phase, these rules are evaluated for iterations

$i = 0$ until no more δ^+ are derived. The put-back phase produces the correct tuples into `reachable`.

4.3 Provenance-based Incremental Maintenance

Despite of its guarantee to terminate on the maintenance of recursive rules, the DRed method removes a lot of tuples redundantly, only to rederive them again. The example of retracting `link(c,c)` illustrates this point exactly. Since node `d` is still reachable from node `a` (Figure 1.1), all of the deleted `reachable` tuples (with the exception of `reachable(c,c)`) will be inserted back in the put-back phase.

As an alternative to DRed, the Orchestra [50] system proposed the use of provenance information to perform a derivability test to determine whether tuples ought to be deleted in the presence of base tuple deletions. This has been further refined in [87], which uses a compact form of *absorption provenance*, which enables us to directly detect when view tuples are no longer derivable and should be removed. For example, assuming we annotate the base tuples `link(a,b)`, `link(b,c)`, `link(c,a)`, and `link(c,b)` with $p1$, $p2$, $p3$, and $p4$ respectively. The provenance expression for `reachable(b,b)` is $(p_2 \wedge p_4) \vee (p_1 \wedge p_2 \wedge p_3)$, given that there are two alternative derivations for `reachable(b,b)`.

Absorption provenance shows its value in handling deletions. When $link(c, b)$ is deleted, the only step required with absorption provenance is to zero out p_4 in the provenance expressions of all *reachable* tuples. In this example, zeroing out this derivation will result in a new provenance expression $(p_1 \wedge p_2 \wedge p_3)$, which means that `reachable(b,b)` is still derivable. Encoding for absorption provenance can be achieved efficiently via the use of the *binary decision diagram* [30] (BDD) data structures.

4.4 Incremental Maintenance for Negation and Aggregates

In order to compute Datalog rules with aggregation, incremental fixpoint evaluation techniques [97] have been proposed. These techniques are amenable to incremental pipelined query processing. They are generally applied to *monotonic aggregates* [106] such as `min`, `max` and `count` incrementally, based on the current aggregate and each new input tuple. Programs with stratified aggregation can be maintained using standard algorithms for incremental maintenance of non-recursive aggregate SQL queries, by (1) computing the sets of insertions and deletions to the predicates occurring in the bodies of aggregate rules, then (2) applying those algorithms, viewing both IDB and EDB predicates as source tables.

Programs with stratified negation can be maintained using standard algorithms for incremental maintenance as well [52], where (1) a negated predicate $\neg q$ can be computed from q and the particular bindings provided by the positive predicates in the rule body, and (2) a negated Δ-predicate $\Delta(\neg q)$ is computed directly from Δq and q.

Dong et al. [42, 41] presents the use of first-order (FO) queries for executing recursive view maintenance, applicable to transitive closure computations of graphs. The practical implication of FO evaluation is that the maintenance can be supported by existing database systems. In particular, [41] presents an incremental evaluation system that uses standard SQL queries executable in a commercial database engine, for supporting transitive closure maintenance of graphs.

5

Datalog Extensions

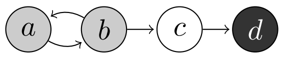
This chapter presents language extensions to the core Datalog introduced earlier in Chapter 2. Some of these extensions are used in applications surveyed in Chapter 6, while others are included since they are of general interest. They include advanced forms of *negation* and *aggregation* (Section 5.1 and 5.2), *arithmetic* (Section 5.3), and *functors* (Section 5.4). We close with a discussion of extensions to incorporate *updates* (Section 5.5).

5.1 Beyond Stratified Negation

While stratified negation as presented in Section 2.3 has a clear and straightforward semantics, it rules out some interesting examples involving recursion through negation. A large number of alternative semantics have been explored to relax this restriction, including negation as failure [34], the stable model semantics [48], the well-founded semantics [121, 47], modular stratification [104, 61], and universal constraint stratification [103, 105]. A proper examination of all these alternatives would require a lengthy survey paper of its own. Here, we briefly present one popular alternative from the list, the *well-founded semantics*.

The well-founded semantics gives meaning to Datalog programs making arbitrary use of negation via a *three-valued model* in which facts in the query output may be either true, false, or undefined. For stratified programs, the semantics agrees with the one given in the previous section, and undefined values do not occur. But with recursion through aggregation, undefined values may be used to resolve certain logical paradoxes. For instance, recall the non-stratified program of Example 2.6. We observed earlier that to satisfy the rules of the program read as logical constraints, either p or q must be present in any model of the program, yet we had no reason to prefer one over the other. In this case, the well-founded semantics assigns both p and q the value undefined.

Another famous example involves the so-called *win-move game* [121, 47]. Here, we have a single EDB relation move representing the legal moves of a game played by two players, player I and player II. The two players take turns moving a pebble on the move graph. A player loses the game if she must play from a position with no moves (i.e., from a node with no outgoing edges). For instance, consider the following move graph:



In this graph, node d is a lost position because it has no outgoing moves. This is indicated by the black shading. Node c, on the other hand, is a won position, because a player can move the pebble from c to d, forcing the other player to lose. This is indicated by the white shading. Nodes a and b are more interesting: from node b, the move to c is a losing move, because c is a won position for the other player; hence the best strategy is to move to a, from which the only choice for the other player is to move back to b. This can go on *ad infinitum*, hence nodes a and b are drawn positions, indicated by the grey shading.

The won, lost, and drawn positions can be computed under the well-founded semantics using a non-stratified Datalog$^\neg$ program consisting of a single rule:

```
win(X) :- move(X,Y), not win(Y).
```

The won, lost, and drawn positions on the graph correspond precisely to the tuples in `win` with values true, false, and undefined, respectively, in the query output.

Operationally, the result of a Datalog$^\neg$ program under the well-founded semantics can be computed via the so-called *alternating fixpoint* procedure [47]. In brief, the basic idea is to have the computation proceed in stages, with negated atoms in the program evaluated at a given stage using the result of the computation from the *previous* stage. This gives rise to an alternating sequence of computations of *overestimates* and *underestimates* of the set of true facts in the final result. After a number of stages polynomial in the size of the source data, the overestimate and underestimate sets will no longer change. At this point, the tuples from the underestimate are collected as the true facts, those not in the overestimate are the false facts, and the tuples that are in the overestimate but not the underestimate are collected as the undefined facts.

Interestingly, it is known that under the well-founded semantics, the win-move game is a normal form for Datalog$^\neg$ programs, in the sense that any Datalog$^\neg$ program can be rewritten into an equivalent Datalog$^\neg$ program whose only recursive rule has the form

```
win(X̄) :- move(X̄), not win (Ȳ).
```

where $\bar{X}$ and $\bar{Y}$ are tuples of variables. It is also known [45, 46] that any *partial* Datalog$^\neg$ program P (one whose output may contain undefined facts) can be transformed into a *total* Datalog$^\neg$ program P' (no undefined facts), such that P and P' agree on the true facts.

5.2 Beyond Stratified Aggregation

While stratified aggregate programs (see Section 2.4) are a natural class of programs with a clear and intuitive semantics, a number of database transformations of practical interest cannot be expressed in this way. For instance, in declarative networking (Section 6.2) the need arises to compute the costs of shortest routing paths in a network using a Datalog program, where `link` tuples now carry a cost attribute. This can be accomplished using a Datalog program with stratified aggregation:

```
cost(X,Y,C) :- link(X,Y,C).
cost(X,Y,C) :- cost(X,Z,C1), link(Z,Y,C2), C = C1 + C2.
shortest(X,Y,min<C>) :- cost(X,Z,Y,C).
```

Here a fact `cost(a,b,c)` indicates that there is a path from `a` to `b` of cost `c`. Although the program has a well-defined semantics, when the `link` graph contains cycles, `cost` may be an infinite relation; hence a fixpoint computation will not even terminate.

An alternative formulation of the program avoids the problem by using recursion through aggregation:

```
cost(X,Y,C) :- link(X,Y,C).
cost(X,Y,C) :- shortest(X,Z,C1), link(Z,Y,C2), C = C1 + C2.
shortest(X,Y,min<C>) :- cost(X,Z,Y,C).
```

Another example of a query for which the natural formulation in Datalog involves recursion through aggregation is the company controls problem [33, 90]. In this example, an EDB fact `owns(a,b,n)` indicates that company `a` owns fraction `n` of the stock of company `b`, and we say that company `a` controls company `b` if more than half of `b`'s shares are owned by directly by `a`, or by indirectly by companies that `a` controls. We can compute the `controls` relation via the following Datalog program using recursion through aggregation:

```
owns_via(X,X,Y,N) :- owns(X,Y,N).
owns_via(X,Z,Y,N) :- controls(X,Z), owns(Z,Y,N).
total_owns_via(X,Y,sum<N>) :- owns_via(X,_,Y,N).
controls(X,Y) :- total_owns_via(X,Y,S), S > 0.5.
```

To deal with these and other examples, Ross and Sagiv [106] present a generalization of stratified aggregation called *monotonic aggregation*. The basic idea is as follows. Given a Datalog$^\neg$ with aggregates program P, we first partition the IDB predicates of P into connected components according to the precedence graph for P, and process one connected component at a time in topological order. Within a component, recursion through aggregation is allowed, so long as the aggregate functions used in the program are *monotone* in a certain precise sense, and the rules of the program obey certain syntactic conditions. These conditions are somewhat technically involved and we omit them here; see the paper by Ross and Sagiv [106] for details. Under these conditions, it can be shown that the immediate consequence operator T_P as defined in the previous section has a least fixpoint (although additional restrictions are required to ensure that the fixpoint can be reached in finitely many steps). Practical incremental evaluation strategies for implementing monotonic aggregation are addressed in the paper by Ramakrishnan et al. [98].

Alternative proposals to monotonic aggregation include an approach based on the well-founded semantics due to Van Gelder [120] and approaches based on semiring-annotated relations [51, 118]. More recently, Conway et al. introduced Bloom^L language which allows monotonic aggregation and other functions to be written using user-defined lattices [37].

At present, the community seems not to have converged on any one of these proposals as "the" standard semantics for aggregation through recursion, and there seems to be room for new research in this area with the goal of balancing expressiveness, efficiency, and usability for the programmer.

5.3 Arithmetic and Infinite Relations

In practical applications the need often arises to express queries involving arithmetical calculations. For example, consider again the reachability example. We might wish to compute, for each node in the network, all the nodes that are reachable from this node, along with the number of hops from this node. This can be expressed naturally in Datalog extended with + as follows:

```
reachable(X,Y,1) :- link(X,Y).
reachable(X,Y,J) :- link(X,Z,I), link(Z,Y), J = I+1.
```

The intended meaning of the program is clear (and indeed, the model-theoretic, proof-theoretic, and fixpoint-theoretic semantics extend to handle queries with +). However, note that in contrast to "normal" EDB predicates like `link`, the graph of the + function is *infinite*. If the expected constraints hold for the `link` relation—in particular, if `link` is acyclic—the example above will be well-behaved in the sense that a least fixpoint computation will terminate in polynomial time in the size of `link`. However, this will not be the case if `link` contains a cycle; and in general fixpoint evaluation of Datalog programs involving infinite relations take superpolynomial time or may even diverge. (Likewise, the least models of such programs may be superpolynomial or even infinite).

Motivated by these observations, a line of theoretical work initiated by Ramakrishnan et al. [96], and continuing in [63, 65, 107, 62, 35], has investigated basic issues with Datalog programs involving infinite relations, in particular focusing on the problem of query *safety* [96] (aka *finiteness* [62]). The safety problem is to decide whether the result of the a given Datalog program can be guaranteed to be finite even when some source relations are infinite.

In the papers mentioned above, additional restrictions on the infinite source relations are typically assumed in the form of *finiteness constraints* [96]. These are, essentially, a relaxation of the standard notion of functional dependencies [12] in which a subset of a relation's attributes are specified are "finitely determining" other attributes. Formally, a finiteness constraint is of the form $p : V \rightsquigarrow W$, where p is a predicate symbol and V and W are sets of argument positions of p. A (possibly infinite) instance P of p satisfies the constraint if P associates a finite set of "W-values" with each "V-value." For example, if P is intended to encode arithmetical addition, i.e., $P(a, b, c)$ holds precisely when $a+b = c$, then P satisfies $p : \{1, 2\} \rightsquigarrow 3$. If, moreover, the domain of P is assumed to be that of the natural numbers, then $p : 3 \rightsquigarrow \{1, 2\}$ holds as well.

The paper by Kifer [62] includes a thorough overview of known results (see also Cohen et al. [35]). It is known, for example, that safety of *monadic* core Datalog programs—where all IDB predicates are unary—with finiteness constraints is decidable [107], and that a stronger notion of *supersafety* [63, 62] is decidable for arbitrary core Datalog programs with finiteness constraints. On the other hand, decidability of safety for core Datalog programs over infinite relations with finiteness constraints remains open.

5.4 Functors

Many applications require “invention” of new values during the course of query computation. For example, when we present data integration and data exchange (Section 6.3) we will see examples where missing values in a target schema need to be “invented” based on combinations of source values. Likewise, when we present declarative networking (Section 6.2) we will need to perform list concatenation—another form of value “invention”—in order to compute routing paths.

To handle these requirements cleanly, Datalog has been extended with the mechanism used in logic programming for such purposes, namely, *uninterpreted functions* (or *functors*).[1] Specifically we allow the use of function symbols $\mathtt{f}, \mathtt{g}, \mathtt{h}, \ldots$ in the bodies and heads of Datalog programs. These are *uninterpreted functions*, not to be confused with user-defined functions: the result of, say, applying `f` to arguments `(1,2)` is not a number, but the value $\mathtt{f}(1,2)$. More precisely, we assume a *Hilbert interpretation* of function symbols—they are interpreted “as themselves”—just as we have done all along for constant symbols.

For example, in declarative networking (Section 6.2), we need to keep track of routing paths in the network.

```
path(S,D,P) :- link(S,D), P = [S,D].
path(S,D,Z,P) :- link(S,Z), path(Z,D,Z2,P2), P = [S,P2].
```

Here $\mathtt{[}\cdot,\cdot\mathtt{]}$ is an uninterpreted function (functor) used to perform list concatenation.

As another example, in data integration and data exchange (Section 6.3), we often need to “invent” new values corresponding to missing schema elements. Suppose, for instance, that we wish to restructure employee data stored in a single table `empl(name, salary)` to conform to another schema in which the same data is stored in two tables `names(empID, name)` and `salaries(empID, salary)`. To do this we must invent a value for `empID`, such that by joining on `empID` we can recover the original information. This can be accomplished using a functor `f` as follows:

[1]Historically, Datalog was defined as essentially Prolog without functors; so we abuse terminology somewhat by continuing to use the term “Datalog” for the extension considered here.

```
names(f(Name,Salary), Name) :- empl(Name, Salary).
salaries(f(Name,Salary), Salary) :- empl(Name, Salary).
```

It is common to refer to `f` also as a *Skolem function*, where the terminology is borrowed from mathematical logic. Data integration and exchange systems often do not expose Datalog directly, but rather allow specification of constraints between database instances using logical formalisms, typically fragments of first- or second-order logic. For the example above the corresponding first-order constraint (a so-called *tuple-generating dependency*) is

$$\forall x\ \forall y\ (\texttt{empl}(x, y) \rightarrow \exists z\ \texttt{names}(z, x) \wedge \texttt{salaries}(z, y))$$

By *Skolemizing* this constraint, we replace the existentially-quantified variable z with a *Skolem function* f:

$$\forall x\ \forall y\ (\texttt{empl}(x, y) \rightarrow \texttt{names}(f(x, y), x) \wedge \texttt{salaries}(f(x, y), y))$$

Intuitively, f produces a "witness" value for z, given input values x and y.

The extension of Datalog with functor terms is extremely powerful: in fact, the resulting language is Turing-complete. Many practical systems, including P2 [93], RapidNet [101], and LogicBlox [4], put the burden of guaranteeing termination on the developer writing the Datalog code.

An alternative explored in the data exchange community is to identify syntactic conditions under which termination in polynomial time can be guaranteed. We present here one commonly-used condition, called *weak acyclicity* [40, 43], originally developed for chasing with collections of logical dependencies and adapted here to work with Datalog with functor terms. Given a Datalog program with functor terms P, we construct a directed *dependency graph* $G = (V, E)$ of P as follows:

- for each IDB predicate `p` of arity k occurring in P, V contains k vertices $(\texttt{p}, 1), \ldots, (\texttt{p}, k)$;
- there is an edge from $(\texttt{p}, i)$ to $(\texttt{q}, j)$ whenever there is a rule A `:-` $\ldots, B \ldots$ such that A is a `p`-atom with a variable term X at position i and B is a `q`-atom with the same variable X occurring at position j

- there is an edge labeled $\star$ from $(\mathtt{p}, i)$ to $(\mathtt{q}, j)$ whenever there is a rule A `:-` $\ldots, B \ldots$ such that A is a `p`-atom with a functor term at position i containing a variable X as an argument, and B is a `q`-atom with the same variable X at position j

We say that P is *weakly acyclic* if its dependency graph does not have a cycle going through an edge labeled $\star$. Intuitively, weakly acyclic programs do not have any "feedback loops" allowing unbounded generation of new values via functor terms. One can show that if P is weakly acyclic, its least fixpoint can be computed in time polynomial in the size of the source database.

Example 5.1. To illustrate, consider the following recursive version of the names and salaries program presented earlier:

```
names(f(Name,Salary), Name) :- empl(Name, Salary).
salaries(f(Name,Salary), Salary) :- empl(Name, Salary).
empl(Name,Salary) :- names(Id,Name), salaries(Id,Salary).
```

The dependency graph of the program is shown below:

$$\begin{tikzcd}
(\texttt{names}, 2) \arrow[r, bend left] & (\texttt{empl}, 1) \arrow[l] \arrow[r, "\star"] \arrow[dl, "\star"] & (\texttt{salaries}, 1) \\
(\texttt{names}, 1) & (\texttt{empl}, 2) \arrow[l, "\star"'] \arrow[r] \arrow[ur, "\star"] & (\texttt{salaries}, 2) \arrow[l, bend left]
\end{tikzcd}$$

Although the dependency graph is cyclic, observe that there is no cycle through an edge labeled $\star$. Therefore the program is weakly acyclic.

Note that many programs of practical interest, including the shortest paths query presented earlier, are not weakly acyclic. We will revisit that example in Section 6.2.

5.5 States and Updates

A useful feature in data management systems is the use of *event-condition-action* rules, where *events* are used to trigger rule executions to apply *actions* given specific *conditions*. Here, events are triggered by tuple updates or timers, conditions reflect the current state of the database, and actions involve tuple insertion and deletions of tuples or the generation of events. One can view the delta rules presented in Section 3.1 for query processing as an instance of such rules. Event-condition-action rules are also useful in other contexts, e.g. flexible enforcement of user-defined constraints, customizing different forms of cascading updates, etc.

Datalog extensions for updates. As one example Datalog extension to incorporate explicit updates, Abiteboul and Vianu [9, 13] consider an extended language called Datalog$^{\neg\neg}$ that allows negative atoms in the rule heads. This provides the ability to retract a previously derived tuple.

Example 5.2. To illustrates an example Datalog$^{\neg\neg}$ rule, we consider a constraint imposed on table `salaries(empID,salary)` (introduced in Section 5.4): the `salaries` table should not contain information about people who have retired. Assuming table `retire(empID,date)` maintains the retirement dates of former employees, such a constraint can be implemented using the following Datalog$^{\neg\neg}$ rule:

```
not salaries(empID,salary) :- salaries(empID,salary), retire(empID,date).
```

The introduction of deletion in the rule head results in a *non-inflationary* semantics, i.e. an IDB tuple that has been derived previously may be deleted subsequently during the fixpoint computation. For instance, the insertion of `retire` tuples may lead to the deletion of `salaries` tuples, and hence, the eventual fixpoint may end up with a smaller number of `salaries` tuples.

In general, the global semantics of a set of interacting active rules is complicated, since the result is highly dependent on the order in which rules are triggered, particularly in situations when an event can trigger multiple rules at the same time, some of which involve actions that update the conditions in other rules that are fired at the same time.

In the case of Datalog$^{\neg\neg}$, there are multiple reasonable options in defining its semantics. Given an event that can trigger multiple rules, a *deterministic* semantics would involve firing all rules triggered by the event in parallel, buffering up all the actions, and applying them only when all rules have completed their execution. A *non-deterministic* semantics applies one rule at a time, and the order is decided arbitrarily. See Widom and Ceri [124] for more details on active database systems.

Statelog. In order to resolve execution ambiguities in rules with explicit state updates, Ludaesher proposed *Statelog* [82], that augments the Datalog language to incorporate an explicit notation of state. State in this case is timestamped logically using natural numbers. We provide the following formal definition.

Definition 5.1. Statelog uses a a new state constant 0 (denoting the initial state), a unary function symbol +1 (mapping a state to its successor), and a state variable S. The set $\mathbb{S}$ *state terms* is the least set such that

(1) $[0] \in \mathbb{S}$, $[S] \in \mathbb{S}$, and

(2) if $[T] \in \mathbb{S}$, then $[T+1] \in \mathbb{S}$

A *Statelog atom* is of the form $[T]p(x_1, ..., x_n)$, where $[T]$ is a state term, p is an n-ary relation symbol, and $x_1, ..., x_n$ are data constants or variables. A Statelog literal is a Statelog atom or its negation.

A *Statelog rule* is an expression of the form

$$[S+k_0]H \;\texttt{:-}\; [S+k_1]B_1, ..., [S+k_n]B_n$$

where $[S+k_0]H$ is a Statelog atom and $[S+k_1]B_1, ..., [S+k_n]B_n$ are Statelog literals. A *Statelog program* is a finite set of Statelog rules.

The basic idea of the Statelog approach is to use the current state and possibly previous states in order to define a new state or extend the current one. In the first case, a rule defines the changes from the current state to its successor, i.e., an update; in the second case, a rule extends the current state and defines a view (that may be referred to by other rules).

Using Statelog, one can model state changes to each atom over time. In fact, Ludaescher [82] showed that one can implement different Datalog semantics, e.g. stratification semantics (Section 2.3) and well-founded semantics (Section 5.1), emulated as Statelog rules.

6

Applications

We conclude our exposition of Datalog with some example applications. In particular, we discuss the domains of program analysis, declarative networking, data integration and exchange, and enterprise software systems. For each domain, we highlight language extensions, runtime considerations, and use cases. We then briefly survey other applications.

6.1 Program Analysis

Program analysis is a term covering a broad range of analysis: dataflow, control-flow, points-to, source code structure, etc. The results of these analysis are used to optimize programs for performance, to discover bugs, to enforce coding standards, etc. The domain of program analysis is particularly suitable for Datalog, as recursion and non-linear recursion in particular, is pervasive in analysis logic. In this section, we first give readers a taste of program analysis in Datalog with an example from a Java points-to analysis; we then provide an overview of the major works in this area, and discuss two in particular in more details.

Program analysis by example. Points-to analysis determines what heap object a variable can point to. For instance, given the simple Java statement: `Object a = new Object();`, the variable `a` points to the heap object allocated by the call to `new`. Similarly, given an assignment from one variable to another: `a = b;`, variable `a` points to every heap location pointed to by `b`.

The following two rules are part of a points-to analysis for Java. They specify the recursive relationship between the heap objects a variable can point to (stored in the predicate `varPointsTo(Var,HeapObj)`), and the heap objects that the field of an object can point to (stored in the predicate `fieldPointsTo(BaseObj,Field,Obj)`):

```
r1  fieldPointsTo(BaseObj,Field,Obj) :-
      storeField(From,Base,Field), varPointsTo(Base,BaseObj),
      varPointsTo(From,Obj).

r2  varPointsTo(To,Obj) :-
      loadField(Base,Field,To), varPointsTo(Base,BaseObj),
      fieldPointsTo(BaseObj,Field,Obj).
```

Rule *r1* specifies that, given that a reference is assigned to a field, `Base.Field = From`, where reference `Base` points to heap object `BaseObj`, and reference `From` points to heap object `Obj`, then the field `Field` of heap object `BaseObj` points to `Obj`, as well. That is, a field points to heap locations its assigned references point to.

Rule *r2* specifies that, given that a field is assigned to a reference, `To = Base.Field`, where the reference `Base` points to `BaseObj`, and the field `Field` of `BaseObj` points to `Obj`, then reference `To` points to heap object `Obj`, as well. That is, a reference that has been assigned a field, points to the heap locations that the field points to.

Note that `fieldPointsTo` and `varPointsTo` are mutually recursive, yet `varPointsTo` appears twice in the definition of `fieldPointsTo`. That is, these rules are not linearly recursive rules.

Overview of program analysis using Datalog. Using logic programs and database queries to express program analysis has been explored since the early 90's [118, 39]. The first Datalog-based analysis was introduced by Thomas Reps [102]. Reps specified interprocedural dataflow analysis and program slicing [116] in Datalog, and evaluated the analysis using Coral [99]. The goal of Reps' work was to demonstrate that not only can one specify a program analysis in Datalog, a demand-driven variant of the analysis, where answers are computed lazily, rather than exhaustively for the whole program, can be naturally derived by applying magic set rewrites.

Reps' seminal work left two major open questions. First, *can a complete program analysis be specified in Datalog?* Reps' analysis made many simplifying assumptions about the program analyzed. Is Datalog expressive enough for program analysis that can be used in practice? Secondly, *can analysis specified in Datalog scale?* Scalability of program analysis is an issue in general regardless of the language they are implemented in. Is it possible then, for program analysis programs specified in a high level language like Datalog, to match, or even exceed, the performance of their counterparts implemented in imperative languages such as C or Java?

These two questions have been the focus of recent research in Datalog-based program analysis. We next describe two frameworks that distinguish themselves in the completeness of the analysis implemented, and in the extensive evaluations provided for their implementations.

.QL. .QL, originally named CodeQuest [54], is a source code query language originally developed at Oxford University. .QL has since been commercialized by Semmle, Inc. [7].

.QL focuses on supporting queries of static properties of programs: what are the subclasses of class `A`? Do all subclasses of `A` override method `m`? .QL queries are in expressiveness equivalent to globally stratified Datalog 2.3.[1] .QL queries were originally evaluated by translation to SQL queries (with an external fixpoint control), and executed against relational database (supported databases are Microsoft

[1].QL provides an Object-Oriented syntax that makes certain queries more concise, and provide facilities for packaging up groups of queries as libraries. The syntax, however, translates down to plain globally stratified Datalog, and requires no semantic extensions.

SQL Server[5], PostgreSQL[6], and H2[3]). Performance concerns drove Semmle to eventually implement an in-memory Datalog engine.

.QL's goal is to enable Java or C++ programmers to write custom queries to understand their code. To this end, .QL provides a large library of IDB rules, defining predicates that are useful in common code queries. This particular usage scenario posed an interesting optimization challenge: library rules cannot be written to perform well for all queries. Thus, library rules must be optimized in the context of the queries. This challenge drove the development of cost-based magic sets optimizations [108] discussed in Section 3.3.

Doop. Doop[28, 29] is a points-to analysis framework for Java. Doop is implemented in globally stratified Datalog, evaluated on a commercial deductive database engine built by LogicBlox, Inc.[4]. Doop is the only system to specify points-to analysis completely within Datalog. Compare to previous attempts [123, 66], Doop declaratively specifies on-the-fly call-graph construction, handles Java language features that are crucial for the completeness of the analysis, such as reflection, native code, finalization.

Doop is the first declarative program analysis system shown to *outperform* hand-tuned program analysis written in imperative languages, e.g. Java. Compared to PADDLE[70], the prior art program analysis for Java, written in Java, Doop achieved over 10x speed-up on average, for logically equivalent analysis. Doop is also the only analysis framework to be able to scale up to sophisticated analysis, such as the 2-call-site-sensitive analysis with a context-sensitive heap [29].

Perhaps surprisingly, Doop achieves its performance through no use of optimization techniques that are particular to Datalog, such as magic sets. Its performance is derived purely from applications of traditional query optimization techniques: the use of alternative indices, and folding of commonly used sub-queries.

6.2 Declarative Networking

Declarative networking [76, 80, 77, 75] is a programming methodology that enables developers to concisely specify network protocols and services using a distributed recursive query language, and directly compile these specifications into a dataflow framework for execution. This approach provides ease and compactness of specification, and other additional benefits such as optimizability and the potential for safety checks.

As evidence of its widespread applicability, declarative networking techniques have been used in several domains including fault tolerance protocols, cloud computing, sensor networks, overlay network compositions, anonymity systems, mobile ad-hoc networks, secure networks, network configuration management, network forensics, optimizations, and as a basis for course projects in a distributed systems class. There are currently a number of open-source implementations of declarative networking, for instance, P2 [93] and RapidNet [101]. See Loo et.al. [78] for a survey of recent use cases.

6.2.1 Network Datalog

We introduce the *Network Datalog* (*NDlog*) language used in declarative networking with an example program shown below that implements the *Path-vector protocol*, which computes in a distributed fashion, for every node, the shortest paths to all other nodes in a network. The path-vector protocol is used as the base routing protocol for exchanging routes among Internet Service

```
sp1 path(@Src,Dest,Path,Cost) :- link(@Src,Dest,Cost), Path = [Src, Dest].
sp2 path(@Src,Dest,Path,Cost) :- link(@Src,Nxt,Cost1),
    path(@Nxt,Dest,Path2,Cost2), Cost=Cost1+Cost2, Path = [Src, Path2].
sp3 spCost(@Src,Dest,min<Cost>) :- path(@Src,Dest,Path,Cost).
sp4 shortestPath(@Src,Dest,Path,Cost) :- spCost(@Src,Dest,Cost),
    path(@Src,Dest,Path,Cost).
query(@Src,Dest,Path,Cost) :- shortestPath(@Src,Dest,Path,Cost).
```

The program has four rules (which for convenience we label `sp1-sp4`), and takes as input a base (*extensional*) relation `link(Src, Dest, Cost)`. Rules `sp1-sp2` are used to derive "paths" in the graph, represented as tuples in the derived (*intensional*) relation `path(Src,Dest,Path,Cost)`. The `Src` and `Dest` fields represent the source and destination endpoints of the path, and Path is the actual path from Src to node Dest. The number and types of fields in relations are inferred from their (consistent) use in the program's rules.

Since network protocols are typically computations over distributed network state, one of the important requirements of *NDlog* is the ability to support rules that express distributed computations. *NDlog* builds upon traditional Datalog by providing control over the storage location of tuples explicitly in the syntax via *location specifiers*. To illustrate, in the above program, each predicate has an "@" symbol prepended to a single field denoting the location specifier. Each tuple generated is stored at the address determined by its location specifier. For example, each path and link tuple is stored at the address held in its first field @Src.

Rule sp1 produces path tuples directly from existing link tuples, and rule sp2 recursively produces path tuples of increasing cost by matching (joining) the destination fields of existing links to the source fields of previously computed paths. The matching is expressed using the repeated Nxt variable in link(Src,Nxt,Cost1) and path(Nxt,Dest,Path2,Cost2) of rule sp2. Intuitively, rule sp2 says that "if there is a link from node Src to node Nxt, and there is a path from node Nxt to node Dest along a path Path2, then there is a path Path from node Src to node Dest where Path is computed by prepending Src to Path2". The matching of the common Nxt variable in link and path corresponds to a *join* operation used in relational databases.

Given the path relation, rule sp3 derives the relation spCost(Src,Dest,Cost) that computes the minimum cost Cost for each source and destination for all input paths. Rule sp4 takes as input spCost and path tuples and then finds shortestPath(Src,Dest,Path,Cost) tuples that contain the shortest path Path from Src to Dest with cost Cost. Last, the shortestPath table is the output of interest.

6.2.2 Query Evaluation

In declarative networking, each node runs its own set of *NDlog* rules. Typically, these rules are common across all nodes (that is, all nodes run the same protocol), but may further include per-node policy customizations. *NDlog* rules are compiled and executed as *distributed dataflows* by the query processor to implement various network protocols.

To execute *NDlog* programs, declarative networking uses the *pipelined semi-naïve* (PSN) model [75]. PSN extends the traditional *semi-naïve* Datalog evaluation strategy to work in an asynchronous distributed setting. PSN relaxes semi-naïve evaluation to the extreme of processing each tuple as it is received. This provides opportunities for additional optimizations on a per-tuple basis. New tuples that are generated from the semi-naïve rules, as well as tuples received from other nodes, are used immediately to compute new tuples without waiting for the current (local) iteration to complete.

In practice, most network protocols execute over a long period of time and incrementally update and repair routing tables as the underlying network changes (for example, due to link failures, and node departures). Incremental recursive view maintenance techniques [87, 91] provide timely updates and for avoiding the overhead of recomputing all routing tables "from scratch" whenever there are changes to the underlying network.

The *Dedalus* [57, 17] language is similar to *NDlog*, except its behavior and output are defined in terms of a model-theoretic semantics. Dedalus also allows users to write rules that mutate state. The *CALM Conjecture*, posed by Hellerstein [57] states that monotonic *coordination-free* Dedalus programs are *eventually consistent*, and non-monotonic programs are eventually consistent when instrumented with appropriate coordination. Recently, Ameloot et al. explored Hellerstein's CALM conjecture using relational transducers [18].

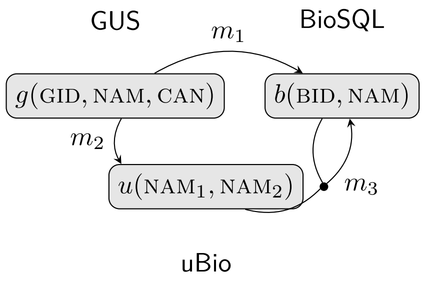
6.3 Data Integration and Exchange

Database management scenarios frequently involve cobbling together heterogeneous data sources or schemas in order to query across them or exchange data among them. This requirement has motivated two closely-related threads of database research in recent years, *data integration* [55, 56] and *data exchange* [95, 43]. In each scenario, we are given a *source* database and schema, a *target* database schema, and a set of *schema mappings* in some logical formalism relating source and target instances. In data integration, the goal is to answer queries posed over the target schema by reformulating them as queries over the source schema. In data exchange, the goal is to materialize a target instance which can be used to answer target queries directly. Both data integration and data exchange also have "peer-to-peer" (PDMS) variants in which the distinction between source and target schemas is relaxed to allow more complex topologies.

Example 6.1. Consider (see Figure 6.1) a bioinformatics collaboration scenario based on databases of interest to affiliates of the Penn Center for Bioinformatics. In general, GUS, the Genomics Unified Schema [53] covers gene expression, protein, and taxon (organism) information; BioSQL, affiliated with the BioPerl project [1], covers very similar concepts; and a third schema, uBio [8], establishes synonyms among taxa. Instances of these databases contain taxon information that is autonomously maintained but of mutual interest to the others. For the purposes of our example, we show only one relational table in each of the schemas, as follows. Peer GUS associates taxon identifiers, scientific names, and what it considers *canonical* scientific names via relation $g(\textsc{gid}, \textsc{nam}, \textsc{can})$; peer BioSQL associates its own taxon identifiers with scientific names via relation $b(\textsc{bid}, \textsc{nam})$; and peer uBio records synonyms of scientific names via relation $u(\textsc{nam}_1, \textsc{nam}_2)$.

The participants of the PDMS collaboration specify the relationships among their databases using *schema mappings*.

Example 6.2. Continuing with Example 6.1, suppose it is agreed in this collaboration that certain data in GUS should also be in BioSQL. This is represented in Figure 6.1 by the arc labeled m_1. The specification $g(G, N, C) \rightarrow \exists B\ b(B, N)$ associated with m_1 is read as follows: if (G, N, C) in table g, the value N must also be in some tuple (B, N) of table b, although the value B in a such a tuple is not determined. The specification just says that there must be such a B and this is represented by the existential quantification $\exists B$. Here m_1 is an example of *schema mapping.* Two other mappings are also shown in Figure 6.1.


GUS
BioSQL
uBio
m_1
m_2
m_3
$g(\text{GID}, \text{NAM}, \text{CAN})$
$b(\text{BID}, \text{NAM})$
$u(\text{NAM}_1, \text{NAM}_2)$

Tgds `Rules.`

(m_1) $\boxed{g(G,N,C) \to \exists B\ b(B,N)}$

```
b(f(N),N) :- g(G,N,C).
```

(m_2) $\boxed{g(G,N,C) \to u(N,C)}$

```
u(N,C) :- g(G,N,C).
```

(m_3) $\boxed{b(B,M) \wedge u(M,N) \to b(B,N)}$

```
b(B,N) :- b(B,M), u(M,N).
```

Figure 6.1: Mappings among three bioinformatics databases. On the right, tgd mappings are shown along with their translations into Datalog rules.

Peer uBio should also have some of GUS's data, as specified by m_2. Mapping m_3 is quite interesting: it stipulates data in BioSQL based on data in uBio but also on data *already* in BioSQL. As seen in m_3, relations from multiple peers may occur on either side. We also see that individual mappings can be "recursive" and that cycles are allowed in the graph of mappings.

Schema mappings are logical assertions that the data instances at various peers are expected to jointly *satisfy*. We shall see that they correspond to the well-known formalism of *tuple-generating dependencies* (tgds) [26]. We shall also see that, as in data exchange [43], a large class of mapping graph cycles can be handled safely, while certain complex examples cause problems.

Thus, every PDMS specification begins with a collection of peers/participants, each with its own relational schema, and a collection of schema mappings between some of these peers. Like the schemas, the mappings are designed by the participants' administrators. By joining the collaboration, the participants agree to share the data from their local databases. The sharing can be further modulated through the mappings, which should therefore be subject to agreement between participants.

Given a PDMS configuration of peers and schema mappings, the question arises of how data should be propagated using the mappings, and what should be the answer to a query asked by one of the peers. The whole point of data integration is for such an answer to use data from *all* the peers. In PDMS, we wish to accomplish this by materializing at every peer an instance containing not only the peer's locally contributed data, but also additional facts that *must* be true, given the data at the other peers along with the constraints specified by the mappings. Queries at a peer will be answered using this local materialized instance. However, while the mappings relating the peers tell us which peer instances are together considered "acceptable," they do not fully specify the complete peer instances to materialize.

PDMS follows established practice in data integration, data exchange and incomplete information databases [11, 43] and uses *certain answers* semantics: a tuple is "certain" if it appears in the query answer no matter what data instances (satisfying the mappings) we apply the query to. In virtual data integration, the certain answers to a query are computed by reformulating the query across all peer instances using the mappings, and combining the answers together from the local results computed at each peer. As in data exchange, PDMS materializes special local instances that can be used to compute the certain answers. This makes query evaluation a fast, local process.

We illustrate this with our running example.

Example 6.3. Suppose the contents of g, u, and b are as shown in Figure 6.2b. Note that the mappings of Figure 6.1 are not satisfied: for example, g contains a tuple

$$(828917, \text{“}\textit{Oscinella frit}\text{”}, \text{“}\textit{Drosophila melanogaster}\text{”})$$

but b does not contain any tuple with "*Oscinella frit*" which is a violation of m_1. We patch this by adding to b a tuple $(\perp_1, \text{“}\textit{Oscinella frit}\text{”})$ where $\perp_1$ represents the unknown value specified by $\exists b$ in the mapping m_1. We call $\perp_1$ a *labeled null.* Adding just enough patches to eliminate all violations results in the data in Figure 6.2c. (Note that sometimes patching one violation may introduce a new violation, which in turn must be patched; hence this process is generally iterative, however under certain constraints it always terminates.) Observe that we can replace $\perp_1$ and the other labeled nulls with combinations of arbi-

GID	NAM	CAN
828917	*Oscinella frit*	*Drosophila melanogaster*
2616529	*Musca domestica*	*Musca domestica*

(a) Table from GUS used in data exchange

NAM_1	NAM_2

BID	NAM
4472	*Periplaneta americana*

(b) Tables from uBio (left) and BioSQL (right) before data exchange

NAM_1	NAM_2
Oscinella frit	*Drosophila melanogaster*
Musca domestica	*Musca domestica*

BID	NAM
4472	*Periplaneta americana*
$\perp_1$	*Oscinella frit*
$\perp_1$	*Drosophila melanogaster*
$\perp_2$	*Musca domestica*

(c) Updated tables after data exchange. Newly-inserted tuples are shaded.

Figure 6.2: Bioinformatics data exchange example

trary values and we get an entire class of data instances that satisfy the mappings.

Now, consider two Datalog queries:

```
p(B,N) :- b(B,N), N = "Oscinella frit".
q(N) :- b(B,N), N = "Oscinella frit".
```

and let us see which answers are *certain* when we apply these queries to the data instances obtained by replacing the labeled nulls with various values. There is no tuple in common among the various answers `p` produces, hence its certain answer semantics is empty. However, all answers produced by `q` have the tuple ("*Oscinella frit*") in common. This is a certain answer for `q`.

The procedure we used in the example to resolve mapping violations by patching instances is intuitive but it is not clear that (1) it always works, and (2) the data instances obtained by replacing labeled nulls with arbitrary values are representative of *all* instances satisfying the mappings and therefore give us the certain answers. In fact, the theory of *data exchange* [43] has resolved both these problems. Moreover, it has established the following convenient query answering algorithm: to obtain the certain answers to a query it suffices to evaluate the query over a specifically computed data instance with labeled nulls (as if the labeled nulls were ordinary values) and then to discard any tuples in the result containing labeled nulls. As a consequence, PDMS systems such as Orchestra [92] works with peer instances in which tuples may contain labeled nulls (actually, in the slightly more complicated form of *skolem functions*, as described in Chapter 5.4.)

6.4 Enterprise Software

The modern enterprise software stack—a collection of applications supporting bookkeeping, analytics, planning, and forecasting for enterprise data—is facing a variety of challenges for its increasing complexity: the task of building and maintaining enterprise software is tedious and laborious; applications are cumbersome for end-users; and adapting to new computing hardware and infrastructures is difficult. Among others, the LogicBlox platform [4] unifies the programming model for enterprise software development that combines transactions with analytics, by using a *declarative* language amenable to efficient evaluation schemes, automatic parallelizations, and transactional semantics.

Next, we present an overview of the DatalogLB language used by the LogicBlox platform, emphasizing extensions to support general-purpose programming and the development of various components of enterprise applications.

Rules. DatalogLB rules are specified using a `<-` notation (instead of the traditional "`:-`"), as in the example below:

```
person(X) <- father(X,Y).
person(X) <- mother(X,Y).
grandfather(X,Z) <- father(X,Y), father(Y,Z) ; father(X,Y), mother(Y,Z).
mother(X) <- parent(X,Y), !father(X).
```

In this example, `;` indicates disjunction while `!` is used for negation. The first two rules copy data from the `father` and `mother` predicates into `person`. The third rule computes the `grandfather` predicate, essentially as the union of two conjunctive queries. Finally, the fourth rule specifies that all parents that are not fathers are mothers, with negation interpreted under the stratified semantics.

Entity types and constraints. The main building-blocks of the DatalogLB type system are *entities*, i.e., specially declared unary predicates corresponding to some concrete object or abstract concept. The DatalogLB type system also includes various primitive types (e.g., numeric types, strings etc). For example, the following DatalogLB program declares (using a `->` notation) that `person` is an entity:

```
person(X) -> .
```

Entities can have various properties, expressed through predicates with the corresponding entity as the type of some argument, for example:

```
ssn[X] = Y -> person(X), int[32](Y).
name[X] = N -> person(X), string(N).
```

The first declaration says that `ssn` is a functional predicate mapping `person` entities to integer-valued Social Security Numbers, while the second maps `person` entities to string names.

Entities can be arranged in subtyping hierarchies, e.g., the following example declares that `male` is a subtype of `person`:

```
male(X) -> person(X).
```

As expected, subtypes inherit the properties of their supertypes and can be used wherever instances of their supertypes are allowed by the type system. For example, according to the declarations above, a `male` also has an `ssn` and a `name`.

One can also use the `->` notation to specify runtime *integrity constraints*, such as that every `parent` relationship is also either a `father` or `mother` relationship, but not both:

```
parent(X,Y) -> father(X,Y), !mother(X,Y) ; mother(X,Y), !father(X,Y).
```

Updates and events. The needs of interactive applications motivate procedural features in DatalogLB (inspired by previous work on Datalog with updates [13] and states [82] presented in Section 5.5). For instance, LogicBlox provides a framework for user interface (UI) programming that allows the implementation of UIs over stored data through DatalogLB rules. Apart from being able to populate the UI based on results of DatalogLB programs, UI events are also handled through DatalogLB rules that are executed in response to those events.

Example 6.4. Consider a simple application in which managers are allowed to use a form to modify sales data for planning scenarios. This form, including the title of the page, the values in a drop-down menu and the text on a "submit" button, is generated by the DatalogLB rules shown below.

```
sales_entry_form(F) -> form(F).

form_title[F] = "Sales Data Entry"
   <- sales_entry_form(F).

component[F] = D, dropdown(D), label[D] = "item"
   <- sales_entry_form(F).

submit_button[F] = B, label[B] = "submit"
   <- sales_entry_form(F).
```

The selection of values for particular items from the drop-down menu, specifying a UI view, also corresponds to a database view:

```
dropdown_values(D,I)
   <- component[F] = D, sales_entry_form_user(F,U), modifiable_by(I,U).
```

UI events, such as when a submit button is pushed, are represented as predicates, and one can write rules—such as the one below—that are executed when these events happen:

```
^sales[P,D,S] = V
   <- +button_clicked(F,S),
      sales_entry_form_user(F,U), dropdown_selected[F] = P,
      date_fld_value[F,_] = D, num_fld_value[F,_] = V, manager(S,U).
```

This is an example of what LogicBlox terms a *delta rule*[2], used to insert data into the EDB predicate `sales`. In this body, the atom `button_clicked(F,S)` is preceded by the *insert* modifier "`+`", which indicates an insertion to the corresponding predicate. As a result, the rule will only be fired when the submit button is pushed and the corresponding fact is inserted in the `button_clicked` predicate. Similarly, the symbol "`^`" in the head is the *upsert*[3] modifier, indicating that if

[2]This should not to be confused with the delta rules transformation used in semi-naive evaluation.

[3]A combination of update and insert.

the corresponding key already exists in `sales`, its value should be updated to the one produced by the rule, otherwise a new entry with this key-value pair should be inserted.

Constructors. DatalogLB also allows the invention of new values during program execution through the use of *constructors* (aka *Skolem functions*) in the heads of rules. DatalogLB programs using recursion through constructors are not guaranteed to terminate on all inputs. For this reason, the DatalogLB compiler implements a safety check that exploits the connection between Datalog evaluation and the chase procedure [86], and warns if termination cannot be guaranteed. (The same safety check is used for programs using recursion through arithmetic.)

6.5 Other Applications

In addition to the above applications, we briefly survey other recent use cases of Datalog in the domains of security, web data extraction, concurrent programming, and answer-set programming.

6.5.1 Security

Secure Network Datalog (SeNDlog) language [125] unifies *NDlog* and logic-based languages for access control in distributed systems. SeNDlog allows users to specify and implement distributed systems and their security policies within a common declarative framework. The SecureBlox [85] platform, developed in the LogicBlox [4] system, further provides a richer set of features compared to SeNDlog: SecureBlox supports 1) user-defined security constructs that can be customized and composed in a declarative fashion, 2) *meta-rules* – Datalog rules that operate on the rules of the program as input, and produce new rules as output, and 3) *meta-constraints* – Datalog constraints that restrict the allowable rules in the program. SecureBlox allows meta-programmability for compile-time code generation based on the security requirements and trust policies of the deployed environment.

6.5.2 Web data extraction

Web data extraction tools are used to aggregate information relevant to a particular topic from different websites; the extracted information is then presented to users in a single view. For instance, an aggregator of financial news might use a web data extraction tool to collect financial news from several major newspapers; an aggregator of product information may crawl several e-retailers to collect the prices on the same product. Extracting data from (HTML) webpages can be thought as evaluating queries over an XML tree. A query may specify the types of nodes of interest (e.g. tables, italics fonts), and place conditions on the shape and contents of the nodes' children or siblings. The result of the query may be either text or entire XML subtrees.

Lixto [49] commercializes a tool for specifying extractions of web data. Elog, Lixto's query language, is an extension of monadic Datalog—a restricted form of Datalog in which all intensional predicates are unary. Elog queries can be evaluated efficiently, as monadic Datalog over trees has the combined complexity of $O(|P|*|dom|)$, where $|P|$ is the size of the program, and $|dom|$ the size of the tree. Furthermore, extraction queries written in Elog have the benefit of abstracting over changes in the trees that are not relevant to the queries. This results in extractions that are more robust against often frequent changes in the format of HTML pages on different websites.

6.5.3 Concurrent programming

Reactors [44] extend Datalog to provide higher level language abstractions to help programmers cope with the added complexities of programming for concurrency. A reactor is a set of Datalog-like rules, with additional constructs for identifying an event and its associated data, referring to the state of the world before the event, and specifying the updates to the world as a result of the event. These constructs together support both asynchronous and synchronous composition of reactors. Reactors bear resemblance to ECA, and similarly, to Statelog [83, 84], with which ECA can be formalized with.

6.5.4 Answer-set programming

Answer-set programming is a form of logic programming that is particularly geared towards solving search problems that may have multiple satisfying models, e.g. graph coloring. DLV [69] is an answer-set programming language based on Datalog. DLV allows disjunctions in the head of rules—a crucial language feature that allows the expression of choices that can be made in searching for an answer. DLV allows the expression of problems in the complexity class of Σ_2^p, in finite structures.

Acknowledgements

We would like to thank Chen Chen, Alex Gurney, Yang Li, Dong Lin, Walaa Eldin Moustafa, Anduo Wang, and Zhuoyao Zhang for proof reading this survey paper. We also like to thank the reviewers for their extremely thorough and helpful reviews, and Joseph M. Hellerstein for all his guidance and support throughout the entire process of writing this survey paper. This survey paper is in part supported by NSF grants IIS-0812270, CCF-0820208, CNS-0845552, CNS-1040672, CNS-1117052, and CNS-1218066.

References

[1] *BioPerl, http://bioperl.org.*

[2] *Datomic website, http://www.datomic.com/.*

[3] *H2 Database Engine, http://www.h2database.com.*

[4] *LogicBlox website, http://www.logicblox.com/.*

[5] *Microsoft SQL server, http://www.microsoft.com/sql.*

[6] *PostgreSQL, http://www.postgresql.org/.*

[7] *Semmle Web site, http://www.semmle.com.*

[8] *uBio, http://www.ubio.org.*

[9] S. Abiteboul, E. Simon, and V. Vianu. Non-deterministic languages to express deterministic transformations. In *PODS*, 1990.

[10] Serge Abiteboul, Zoe Abrams, Stefan Haar, and Tova Milo. Diagnosis of Asynchronous Discrete Event Systems—Datalog to the Rescue! In *PODS*, 2005.

[11] Serge Abiteboul and Oliver Duschka. Complexity of answering queries using materialized views. In *PODS*, 1998.

[12] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[13] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci.*, 43:62–124, August 1991.

[14] Foto Afrati, Stavros S. Cosmadakis, and Mihalis Yannakakis. On datalog vs. polynomial time. In *PODS*, 1991.

[15] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, 2010.

[16] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, 2011.

[17] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell C Sears. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[18] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. Relational Transducers for Declarative Networking. In *PODS*, 2011.

[19] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. pages 89–148, 1988.

[20] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. The deductive database system LDL++. *TPLP*, 3(1):61–94, 2003.

[21] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), 1987.

[22] Francois Bancilhon. Naive evaluation of recursively defined relations. *On Knowledge Base Management Systems: Integrating AI and DB Technologies*, 1986.

[23] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. *SIGMOD Rec.*, 15(2):16–52, 1986.

[24] BDD-Based Deductive DataBase. `http://bddbddb.sourceforge.net/`.

[25] Catriel. Beeri and Raghu. Ramakrishnan. On the power of magic. In *PODS*, 1987.

[26] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *J. ACM*, 31(4):718–741, 1984.

[27] Nicole Bidoit. *Bases de Données Déductives: Présentation de Datalog.* Armand Colin, 1992.

[28] Martin Bravenboer and Yannis Smaragdakis. *Doop website, http://doop.program-analysis.org/.*

[29] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.

[30] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[31] Dario Campagna, Beata Sarna-Starosta, and Tom Schrijvers. Optimizing Inequality Joins in Datalog with Approximated Constraint Propagation. In Claudio Russo and Neng-Fa Zhou, editors, *Practical Aspects of Declarative Languages, 14th International Symposium, Proceedings.* Springer, 2012.

[32] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE TKDE*, 1(1):146–166, 1989.

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

[34] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[35] Sara Cohen, Joseph Gil, and Evelina Zarivach. Datalog programs over infinite databases, revisited. In *DBPL*, 2007.

[36] Robert M. Colomb. *Deductive Databases and their Applications.* Taylor and Francis, 1998.

[37] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *SoCC*, 2012.

[38] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[39] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *PLDI*, 1996.

[40] Alin Deutsch and Val Tannen. Reformulation of xml queries and constraints. In *ICDT*, pages 225–241, 2003.

[41] Guozhu Dong, Leonid Libkin, Jianwen Su, and Limsoon Wong. Maintaining transitive closure of graphs in sql. *In Int. J. Information Technology*, 5, 1999.

[42] Guozhu Dong and Jianwen Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Inf. Comput.*, 120(1):101–106, 1995.

[43] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *TCS*, 336(1):89–124, 2005.

[44] John Field, Maria-Cristina Marinescu, and Christian Stefansen. Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. *Theor. Comput. Sci.*, 410(2-3):168–201, 2009.

[45] Jörg Flum, Max Kubierschky, and Bertram Ludäscher. Total and partial well-founded datalog coincide. In *ICDT*, 1997.

[46] Jörg Flum, Max Kubierschky, and Bertram Ludäscher. Games and total datalog$^{\neg}$ queries. *Theoretical Computer Science*, 239(2):257–276, 2000.

[47] Allen Van Gelder. The alternating fixpoint of logic programs with negation. *JCSS*, 47(1):185 – 221, 1993.

[48] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[49] Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog, and Sergio Flesca. The Lixto data extraction project: back and forth between theory and practice. In *PODS*, 2004.

[50] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.

[51] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, 2007.

[52] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[53] GUS: The Genomics Unified Schema. `http://www.gusdb.org/`.

[54] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In David Thomas, editor, *ECOOP*, 2006.

[55] Alon Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[56] Y. Halevy, G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation for large-scale semantic data sharing. *VLDB Journal*, 14(1):68–83, 2005.

[57] Joseph M. Hellerstein. Declarative imperative: Experiences and conjectures in distributed logic. 2010. SIGMOD Record 39(1).

[58] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1-3):86–104, 1986.

[59] IRIS (Integrated Rule Inference System) Reasoner. `http://www.iris-reasoner.org/`.

[60] Trevor Jim. SD3: A Trust Management System With Certified Evaluation. In *IEEE Symposium on Security and Privacy*, May 2001.

[61] David B. Kemp. Efficient recursive aggregation and negation in deductive databases. *TKDE*, 10(5), 1998.

[62] Michael Kifer. On the decidability and axiomatization of query finiteness in deductive databases. *JACM*, 45(4):588–633, July 1998.

[63] Michael Kifer, Raghu Ramakrishnan, and Abraham Silberschatz. An axiomatic approach to deciding query safety in deductive databases. In *PODS*, 1988.

[64] Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.

[65] Ravi Krishnamurthy, Raghu Ramakrishnan, and Oded Shmueli. A framework for testing safety and effective computability of extended datalog. In *SIGMOD*, 1988.

[66] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS*, 2005.

[67] Laurent Vieille. Recursive Axioms in Deductive Database: The Query-Subquery Approach. In *1st International Conference on Expert Database Systems*, 1986.

[68] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.

[69] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, July 2006.

[70] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[71] Senlin Liang and Michael Kifer. Deriving predicate statistics in datalog. In *PPDP*, 2010.

[72] Leonid Libkin. *Elements Of Finite Model Theory*. Springer, 2004.

[73] Changbin Liu, Lu Ren, Boon Thau Loo, Yun Mao, and Prithwish Basu. Cologne: A declarative distributed constraint optimization platform. In *VLDB*, 2012.

[74] A. Livchak. Languages for polynomial-time queries. *Computer-based modeling and optimization of heat-power and electrochemical objects*, 1992. In Russian.

[75] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, 2006.

[76] Boon Thau Loo, Tyson Condie, Minos 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.

[77] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[78] 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 *Fourteenth International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2012.

[79] Boon Thau Loo, Joseph M. Hellerstein, Ryan Huebsch, Timo thy Roscoe, and Ion Stoica. Analyzing P2P Overlays with Recursive Queries. Technical Report UCB-CS-04-1301, UC Berkeley, 2004.

[80] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM*, 2005.

[81] Boon Thau Loo, Sailesh Krishnamurthy, and Owen Cooper. Distributed Web Crawling over DHTs. Technical Report UCB-CS-04-1305, UC Berkeley, 2004.

[82] Bertram Ludäscher. *Integration of Active and Deductive Database Rules*, volume 45 of *DISDBIS*. Infix Verlag, St. Augustin, Germany, 1998. PhD thesis.

[83] Bertram Ludäscher, Ulrich Hamann, and Georg Lausen. A logical framework for active rules. In *COMAD*, 1995.

[84] Bertram Ludäscher, Wolfgang May, and Georg Lausen. Nested transactions in a logical language for active rules. In *LID*, 1996.

[85] William R. Marczak, Shan Shan Huang, Martin Bravenboer, Micah Sherr, Boon Thau Loo, and Molham Aref. Secureblox: customizable secure distributed data processing. In *SIGMOD*, 2010.

[86] Michael Meier, Michael Schmidt, and Georg Lausen. On chase termination beyond stratification. *PVLDB*, 2(1):970–981, 2009.

[87] Mengmeng Liu and Nicholas Taylor and Wenchao Zhou and Zachary Ives and Boon Thau Loo. Recursive Computation of Regions and Connectivity in Networks. In *ICDE*, 2009.

[88] Jack Minker. Logic and databases: A 20 year retrospective. In Dino Pedreschi and Carlo Zaniolo, editors, *Logic in Databases*, volume 1154 of *Lecture Notes in Computer Science*, pages 1–57. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0031734.

[89] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. In *SIGMOD*, 1994.

[90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, 1990.

[91] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. Maintaining distributed logic programs incrementally. In *13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2011.

[92] Orchestra Collaborative Data Sharing System. `http://code.google.com/p/penn-orchestra/`.

[93] P2: Declarative Networking System. `http://p2.cs.berkeley.edu`.

[94] Christos H. Papadimitriou. A note on the expressive power of prolog. *Bulletin of the EATCS*, 26:21–22, 1985.

[95] Lucian Popa, Yannis Velegrakis, Mauricio A. Hernández, Renée J. Miller, and Ronald Fagin. Translating web data. In *VLDB*, 2002.

[96] R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of recursive horn clauses with infinite relations. In *PODS*, 1987.

[97] Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient Incremental Evaluation of Queries with Aggregation. In *SIGMOD*, pages 204–218, 1992.

[98] Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *SIGMOD*, 1994.

[99] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL deductive system. *VLDB Journal*, 3(2):161–210, 1994.

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

[101] RapidNet Declarative Networking Engine. `http://netdb.cis.upenn.edu/rapidnet/`.

[102] Thomas Reps. Demand interprocedural program analysis using logic databases. *Applications of Logic Databases*, pages 163–196, 1994.

[103] Kenneth Ross. A syntactic stratification condition using constraints. In *ILPS*, 1994.

[104] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. *J. ACM*, 41:1216–1266, November 1994.

[105] Kenneth A. Ross. Structural totality and constraint stratification. In *PODS*, 1995.

[106] Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.

[107] Y. Sagiv and M. Y. Vardi. Safety of datalog queries over infinite databases. In *PODS*, 1989.

[108] Damien Sereni, Pavel Avgustinov, and Oege de Moor. Adding magic to an optimising datalog compiler. In *SIGMOD*, 2008.

[109] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*, 1996.

[110] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[111] Divesh Srivastava and Raghu Ramakrishnan. Pushing constraint selections. In *PODS*, 1992.

[112] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, 2nd edition, 1994.

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

[114] Peter J. Stuckey and S. Sudarshan. Compiling query constraints (extended abstract). In *PODS*, 1994.

[115] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *VLDB*, 1991.

[116] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[117] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10:289–321, September 1985.

[118] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies.* W. H. Freeman & Co., New York, NY, USA, 1990.

[119] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23:733–742, October 1976.

[120] Allen Van Gelder. The well-founded semantics of aggregation. In *PODS*, 1992.

[121] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:619–649, July 1991.

[122] Moshe Y. Vardi. The complexity of relational query languages. In *STOC*, 1982.

[123] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[124] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan-Kaufmann, 1996.

[125] Wenchao Zhou, Yun Mao, Boon Thau Loo, and Martín Abadi. Unified Declarative Platform for Secure Networked Information Systems. In *ICDE*, 2009.