Modern Datalog Engines

Bas Ketsman[1] and Paraschos Koutris[2]

[1] *Vrije Universiteit Brussel, Belgium; bas.ketsman@vub.be*
[2] *University of Wisconsin-Madison, USA; paris@cs.wisc.edu*

ABSTRACT

Recent years have seen a resurgence of interest from both the industry and research community in Datalog. Datalog is a declarative query language that extends relational algebra with recursion. It has been used to express a wide spectrum of modern data management tasks, such as data integration, declarative networking, graph analysis, business analytics, and program analysis. The result of this long line of research is a plethora of Datalog engines, which support different variants of Datalog, and have different technical specifications and capabilities. In this monograph, we provide an overview of the architecture and technical characteristics of these Datalog engines. We identify common architectural decisions and evaluation methods, as well as data structures and layouts used to speed up the query execution. We also discuss in what ways Datalog engines differ when they specialize to workloads with different characteristics (for example, data analytics vs program analysis vs graph analysis). One particular focus is how modern Datalog engines scale to massively parallel environments.

Bas Ketsman and Paraschos Koutris (2022), "Modern Datalog Engines", Foundations and Trends® in Databases: Vol. 12, No. 1, pp 1–68. DOI: 10.1561/1900000073.
©2022 B. Ketsman and P. Koutris

1

Introduction

Recent years have seen a resurgence of interest from both the industry and research community in Datalog. Datalog is a declarative query language that extends Relational Algebra with recursion. It offers a simple declarative interface to the developer, while allowing for general optimizations that can speed up evaluation both in single-machine and parallel settings. During the past two decades, Datalog has been used to express a wide spectrum of tasks in different application domains, such as data integration (Fagin *et al.*, 2003), declarative networking (Loo *et al.*, 2006), graph analysis (Seo *et al.*, 2015; Shkapsky *et al.*, 2016), business analytics (Aref *et al.*, 2015), program analysis (Whaley and Lam, 2004; Smaragdakis and Balatsouras, 2015), and security (Marczak *et al.*, 2010).

Datalog received a lot of academic interest during the late 1980s and early 1990s. During this time, the theoretical background of Datalog was firmly established, including its syntax, semantics, and evaluation methods. Moreover, several mature Datalog systems were developed, among them Coral (Ramakrishnan *et al.*, 1993), LDL (Chimenti *et al.*, 1990), and Glue-Nail (Derr *et al.*, 1994). However, these systems were not widely adopted and their development ceased.

This trend was reversed during the last decade, with the development and deployment of several Datalog engines, both from academia and industry. The list of systems includes LogicBlox (Aref *et al.*, 2015), BigDatalog (Shkapsky *et al.*, 2016), SociaLite (Seo *et al.*, 2015), bddbddb (Whaley and Lam, 2004), Soufflé (Scholz *et al.*, 2016), RecStep (Fan *et al.*, 2019), and Graspan (Wang *et al.*, 2017). In order to deal with the huge volume of data they have to process, the design of these Datalog engines focused around efficiency and scalable performance. Apart from this, their architecture and applications show a lot of variation. Some Datalog systems were developed to target program analysis tasks, a fundamental area in the field of programming languages (e.g., bddbddb, Soufflé). Other Datalog engines specialized on data management problems, such as graph processing (e.g, SociaLite) or business analytics (e.g., LogicBlox). This variation has created a diverse ecosystem of systems that support different variants of Datalog, make different design decisions, and have different capabilities.

In this monograph, we dive deep into this ecosystem and provide an overview of the architecture and technical characteristics of the above Datalog systems. We identify common architectural decisions and evaluation methods, as well as data structures and layouts used to speed up the query execution. We also discuss in what ways Datalog engines differ when they specialize to workloads and inputs with different characteristics (for example, data analytics vs program analysis vs graph analysis). One particular focus of this monograph is how modern Datalog engines scale to massively parallel environments, which is necessary to support the processing of very large datasets. This is a particularly challenging task, since evaluating a Datalog program is generally not an embarrassingly parallel task.

Finally, we remark that Datalog engines have a different focus than SQL engines. While SQL engines are typically treated as one of many interconnected elements in a fully-blown, transactional database management system, Datalog engines are generally more stripped down with a strong focus on efficient recursive query processing. Another difference is that most SQL engines make use of bag-semantics with duplicate elimination as an afterthought, while Datalog engines rely on set-semantics and therefore require specialized data structures and techniques for efficiently dealing with sets throughout the entire computation.

1.1 An Overview of Datalog Systems

Next, we briefly introduce and describe the Datalog systems that will be the focus of this monograph.

LogicBlox (Aref *et al.*, 2015) is a proprietary Datalog system that provides a general platform for developing enterprise software. It has been used as the underlying engine for both points-to program analysis by Doop (Bravenboer and Smaragdakis, 2009), as well as for security applications by SecureBlox (Marczak *et al.*, 2010).

BigDatalog (Shkapsky *et al.*, 2016) is a parallel Datalog engine built on top of a modified version of Apache Spark. Its main applications are business and graph analytics.

SociaLite (Seo *et al.*, 2015) is a parallel Datalog engine used mainly for social network analysis. It is optimized for executing programs on graphs, and has both a single-threaded and parallel implementation.

bddbddb (Whaley and Lam, 2004) is a single-threaded implementation of Datalog designed specifically to support context-sensitive program analysis. Its novelty is the use of binary decision diagrams (BDDs) to compactly represent relations.

Soufflé (Scholz *et al.*, 2016) is a state-of-the-art open-source Datalog system that is used for different types of program analysis, with an emphasis on scalability. It compiles Datalog programs to fast parallel C++ code.

RecStep (Fan *et al.*, 2019) is a multi-threaded implementation of Datalog focused on main-memory execution. It compiles Datalog programs directly to a sequence of SQL queries, which is then executed on Quickstep, a parallel in-memory relational database engine (Patel *et al.*, 2018).

RapidNet (RapidNet, n.d.) is a distributed Datalog engine for the NDLog (Abadi and Loo, 2007) Datalog variant. It compiles Datalog programs directly into C++ programs which run over the NS-3 network simulator (ns-3, n.d.).

Myria and Naiad Finally, we will include in our study two efforts of executing Datalog programs on top of scalable parallel systems: Myria (Halperin *et al.*, 2014) and Naiad (Murray *et al.*, 2013). These do not constitute fully-fledged Datalog implementations, since they require that the user manually specifies an execution plan for the Datalog program they want to run.

In addition to the aforementioned systems, there exist several other Datalog engines: AbcDatalog (Bembenek *et al.*, n.d.), the μZ system built on top of Z3 (Hoder *et al.*, 2011), DES, Differential Datalog (Ryzhyk and Budiu, 2019), the DLV engine (Leone *et al.*, 2006), which supports Disjunctive Datalog, Dyna (Eisner and Filardo, 2011), a system that extends Datalog for AI applications, Bud (Alvaro *et al.*, 2011) and WebdamLog (Moffitt *et al.*, 2015). A few other efforts have focused on adding recursion or fixpoint computation on top of existing relational systems without achieving the expressivity of full Datalog. These include extensions of MapReduce (Afrati *et al.*, 2011; Shaw *et al.*, 2012), or more recent work by Gu *et al.* (2019) and Jachiet *et al.* (2020). Although the above systems will not be the focus of this monograph, we will present some of their techniques that are relevant to recursive computation.

1.2 Commercial Impact

In addition to the success of LogicBlox, several other commercial systems that use Datalog or dialects of Datalog have been developed over the last decade. We discuss next some of these developments.

Datomic is a transactional and distributed database that uses an extended form of Datalog as the basic query language. Semmle, a platform that supports code analysis over large code bases (acquired by Github), has developed its own query language, called .QL, which can be viewed as an object-oriented version of Datalog. Nicira (acquired by

VMware) also uses a restricted variant of the Datalog language, called nLog, for software defined networking. Finally, Datalog was used as the underlying language for declarative analytics in Netsil (acquired by Nutanix).

1.3 Relationship with Prior Work

This survey aims to present how modern large-scale systems evaluate Datalog. It is orthogonal to a rich set of papers and surveys on the Datalog language. What follows is a short but not complete list of relevant work that is complementary to this monograph.

- The most related survey to this monograph is by Green *et al.* (2013). It covers the core Datalog language and its extensions, semantics, query optimizations, incremental view maintenance, along with discussing modern applications of Datalog. Although there is some content overlap with this monograph, our focus is on system building and parallel evaluation. Furthermore, there has been tremendous progress in the adoption of Datalog techniques and many new use cases have been added after its publication.

- Several articles provide a general overview of Datalog and techniques for optimizing recursive computation (Ceri *et al.*, 1989; Ramakrishnan and Ullman, 1995; Bancilhon and Ramakrishnan, 1986). The basic principles of Datalog are also covered in several database textbooks (for example, see Abiteboul *et al.* (1995) and Ullman (1989)).

- A long line of work in the Programming Languages community has also studied the application of Datalog to different types of program analysis tasks (Reps, 1993; Lam *et al.*, 2005). For a comprehensive survey on this subject, we refer the reader to Smaragdakis and Balatsouras (2015).

1.4 Organization

The monograph is organized as follows.

The Datalog Language In Section 2, we introduce the core Datalog language and present its syntax and semantics. Then, we study how different Datalog engines extend the core language to increase its expressibility, with features that are necessary to make it usable in practical settings. These features include negation, (recursive) aggregation, arithmetic operations and functions. We discuss how some Datalog engines choose combinations of features that lead to imprecise language semantics.

Methods for Datalog Evaluation In Section 3, we discuss the fundamental methods used for Datalog evaluation. We focus on the most widely used technique, called *semi-naïve evaluation*, and we discuss the design choices of implementing it. Next, we study how modern Datalog engines use parallelism to further speed up evaluation and why this can be done effectively – both from a theoretical and practical viewpoint. Our focus is both on parallel and multi-threaded environments. We look at the design choices for parallel evaluation across different axes: data partitioning methods, synchronous vs asynchronous evaluation, and data shuffling strategies.

Data Layouts and Indices In Section 4, we study the data layouts used by Datalog engines. Even though the input of Datalog programs is relational data, many systems choose formats other than the traditional row-store: these include tries, binary decision diagrams, bit matrices, and tail-nested tables. These layouts are specialized to be performant for specific inputs and workloads (e.g., graph-oriented data, context-sensitive pointer analysis, dense relations). We also discuss indexing techniques that are specialized and fine-tuned to Datalog evaluation.

Optimization Techniques In Section 5, we present the different optimizations that are used by Datalog engines, both on the language level (e.g., program transformations and rewritings), as well as low-level optimizations on the operator level. Specifically, we study how we can rewrite a Datalog program to push selection and aggregation through recursion, or to reduce the number of iterations and strata.

Finally, we conclude the monograph in Section 6 where we discuss opportunities for future research directions and new possible applications for Datalog engines.

2

The Datalog Language

In this section, we introduce the syntax and semantics of the Datalog language and its extensions. Even though Datalog has been presented in detail in a prior survey by Green *et al.* (2013), we present here its basics in order to make this monograph self-contained. In addition to the core language syntax and semantics, we discuss its extensions that are typically used in modern Datalog engines: these include negation, aggregation (and in particular recursive aggregation), as well as functions and arithmetics (Table 2.1).

Table 2.1: Language Features of Datalog Systems

	negation	**aggregation**	**functions**
BigDatalog	stratified	recursive	yes
Soufflé	stratified	non-recursive	yes
SociaLite	stratified	recursive	yes
RecStep	stratified	recursive	yes
LogicBlox	stratified/general	recursion	yes
bddbddb	stratified	no	yes

2.1 Datalog Basics

We present Datalog (Abiteboul *et al.*, 1995) based on classical predicates under the fixpoint semantics and define the relevant terminology necessary to understand the remainder of this monograph.

2.1.1 Syntax

Datalog programs are constructed from atoms and terms. A *term* is a variable or a constant. We assume throughout this monograph the existence of an infinitely large domain of variables $\mathbf{V}$ and an infinitely large but disjoint domain of data values $\mathbf{D}$ from which variables and constants can be chosen. For convenience of notation, we will consistently choose constants from the range a, b, c, d, e, and variables from the range u, v, w, x, y, z.

An *atom* is a predicate symbol followed by a sequence of terms. An atom without variables is called a *ground atom*. Every predicate symbol has an associated *arity*, referring to the number of terms that it expects. To denote an atom, we write $R(t_1, \ldots, t_m)$ with R the predicate symbol, m the arity of R and $t_1, \ldots, t_m$ a sequence of terms.

A Datalog program is a set of rules of the following form:

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

Here, $n \geq 0$. The atom A is called the *head* of the rule, while the sequence of atoms $B_1, B_2, \ldots, B_n$ is called the *body* of the rule.

The available predicate symbols and their arities are captured in a schema that the program has to be consistent with. Formally, a *schema* $\mathbf{R}$ is a finite set of available predicate symbols R with an associated arity $\mathsf{arity}(R)$. The predicates in a schema $\mathbf{R}$ are divided in two categories[1]: *extensional* (database) predicates (EDBs) are predicates stored in the database and can occur only in the body of rules; *intensional* (database) predicates (IDBs) are predicates defined by rules in the program and can therefore occur in the head as well as body of rules. Among the

[1]The terms extensional and intensional are common in logic and refer to two different approaches of defining things: through explicit listing of their extension (the elements they apply to) or through an intension (definition).

intensional predicates it is common to distinguish between base predicates – predicates that represent base relations in the database – and implicit predicates (including comparisons $=, \neq, \ldots$). Implicit predicates are usually written using infix notation, thus writing for example $x = y$ instead of $=(x, y)$.

We will assume that the rules of a Datalog program are always safe. Formally, a rule is *safe* if all its variables are range-restricted: a variable x is range-restricted if it occurs in the body of the rule in either a (non-negated) atom or in an equality $x = a$ in which a is a constant.

Example 2.1. To demonstrate Datalog syntax, we will not use the textbook example of transitive closure in graphs, but instead we will use an example from the domain of program analysis. Our task is to perform a point-to analysis of variables in a C program (Smaragdakis and Balatsouras, 2015). The instructions of the program can be translated into EDB relations as follows:

AddressOf(a,b) representing `a=&b`
Assign(a,b) representing `a=b`
Load(a,b) representing `a=*b`
Store(a,b) representing `*a=b`

The points-to analysis can now be done using a Datalog program storing pairs of variables a and b in intensional relation PointsTo if a may point to b. The program contains the following four rules with the same head:

```
PointsTo(x,y) :- AddressOf(x,y).
PointsTo(x,y) :- Assign(x,z), PointsTo(z,y).
PointsTo(x,w) :- Load(x,z), PointsTo(z,y), PointsTo(y,w).
PointsTo(x,y) :- Store(z,w), PointsTo(z,x), PointsTo(w,y).
```

In this monograph, we will pay special attention to a class of Datalog programs, called *linear programs*, with favorable computational properties. In a linear program, each rule has at most one IDB relation in its body. In practice, many Datalog programs are linear (or can be made linear).

2.1.2 Semantics

The semantics of positive Datalog can be defined in several equivalent ways. We use in this monograph the least-fixpoint semantics, which defines the outcome of the program in an operational way, based on repeatedly firing the program rules till no more new facts can be derived and thus a fixpoint is reached. It is well-known that this semantics is equivalent to the model-theoretic semantics as well as the proof-theoretic semantics (for more details we refer the reader to Abiteboul *et al.* (1995)).

Datalog programs are computed over instances. A *(database) instance* is a finite set of ground atoms, which we denote by the symbol $\mathbf{I}$. We write $\mathbf{I}(R)$ to denote the relation instance for predicate R induced by $\mathbf{I}$.

The rules of a Datalog program define how new facts are derived from a given instance in terms of variable bindings (also called valuations). A *variable binding* v is a total function from $\mathbf{V}$ to $\mathbf{D}$. For a rule τ, we say that $v(\tau)$ is *satisfied by* instance $\mathbf{I}$ if $v(A) \in \mathbf{I}$, where A is the head atom of τ, or for at least one atom B_i in the body $v(B_i) \notin \mathbf{I}$.

For an instance $\mathbf{I}$ and rule τ, the derived facts are defined by the set

$$\tau(\mathbf{I}) := \{v(A) \mid v \text{ is a variable binding not satisfied by } \mathbf{I}\}.$$

Facts in $\tau(\mathbf{I})$ are called *immediate consequences* of rule τ over instance $\mathbf{I}$. Let $\mathbf{T}_P(\cdot)$ be the operator that adds to instances $\mathbf{I}$ the immediate consequences of all rules in P over $\mathbf{I}$, then the output $P(\mathbf{I})$ of program P over instance $\mathbf{I}$ is the instance obtained by repeatedly applying $\mathbf{T}_P$ (called the *immediate consequence operator* for P) over $\mathbf{I}$ till no more new facts are added; that is, until no more unsatisfied variable bindings for rules in P exist.

The reader may observe that the above fixpoint semantics yields a naive bottom-up evaluation algorithm for computing Datalog programs.

We should note here that Algorithm 1 will always terminate within a polynomial (in the input size) number of steps, assuming the program is fixed.

Algorithm 1: Naive Evaluation of Datalog

```
  Input: $\mathbf{I}$
  Output: $P(\mathbf{I})$
1 $k \leftarrow 0$;
2 $\mathbf{I}_0 \leftarrow \mathbf{I}$;
3 repeat
4     $k \leftarrow k + 1$;
5     $\mathbf{I}_k \leftarrow \mathbf{T}_P(\mathbf{I}_{k-1})$;
6 until $\mathbf{I}_k = \mathbf{I}_{k-1}$;
7 return $\mathbf{I}_k$
```

2.2 Negation

The core fragment of Datalog cannot express non-monotone properties (Afrati *et al.*, 1995). To achieve this expressibility, we need to extend positive Datalog with *negation*. Several options exist to extend positive Datalog with negation. In this monograph, we introduce two commonly used extensions: semi-positive and stratified Datalog. These form the foundation of the languages supported by most modern Datalog execution engines, including SociaLite, Soufflé, LogiQL, etc.

Syntactically, Datalog with negation allows rules to have, besides (positive) atoms, also negated atoms in the body of rules. The use of negation has a safety restriction, which means that every variable in the body of a rule must occur in at least one positive atom. This restriction is necessary to make the output of the program independent of the domain of the attributes.

A Datalog program with negation is *stratified* if the rules of the program can be divided in disjoint subprograms $G_1, G_2, \ldots, G_n$ (henceforth called *strata*) of rules such that the following properties are true for every IDB predicate R:

- All rules defining predicate R occur in the same subprogram, say subprogram G_i;
- All rules mentioning predicate R in a non-negated atom occur in groups G_j, with $j \geq i$; and

- All rules mentioning predicate R in a negated atom occur in groups G_j, with $j > i$.

We note that stratification is not always possible.

A Datalog program with negation is *semi-positive* if all negated atoms have an extensional predicate symbol; that is, if the program is stratified with only one stratum.

Example 2.2. We demonstrate Datalog with negation using an example from graph analytics. We are given a directed graph expressed by the relation Edge(A, B): a tuple Edge(a, b) means that node a has an directed edge to node b. Our goal is to compute all pairs of nodes (a, b) such that there is a directed path from a to b, but not a directed path from b to a. This task can be expressed by the following Datalog program:

```
T(x,y) :- Edge(x,y).
T(x,y) :- T(x,z), Edge(z,y).
P(x,y) :- T(x,y), not T(y,x).
```

This is a Datalog program with stratified negation: the first stratum consists of the first two rules (computing the IDB T), and the second stratum consists of the third rule. It is not semi-positive, since negation occurs in front of the IDB relation T.

The output of semi-positive Datalog programs is defined as before, except that for a rule τ and instance $\mathbf{I}$, $v(\tau)$ is *satisfied by* $\mathbf{I}$ if $v(A) \in \mathbf{I}$, $v(B_i) \notin \mathbf{I}$ for one of the positive atoms B_i in the body, or $v(B_i) \in \mathbf{I}$ for one of the negated atoms B_i in the body of the rule. The output of a stratified Datalog programs over a given database $\mathbf{I}$ is defined as the result of applying the different strata of the program (which can be seen as sub-programs) one-by-one after each other, while taking as input the result of the previous stratum.

2.3 Aggregation

In order to support any type of data analytics, it is necessary to extend Datalog to support some form of aggregation. Many Datalog systems support aggregation in a similar way as they support negation: through stratification. An aggregate rule then has the following form.

$$R(x_1, \ldots, x_n, F(x_0)) \text{ :- } \psi(x_1, \ldots, x_n, x_0).$$

Here, F is an aggregate function, for instance min, max, sum, count, and $\psi(x_1, \ldots, x_n, x_0)$ is some rule body mentioning at least all variables of the head. The following Datalog program is one with stratified aggregation.

Example 2.3. For an example, suppose that we want to compute the connected components in a graph whose relations are represented by the binary relation Edge.

```
       Connected(x,y) :- Edge(x,y).
       Connected(x,y) :- Connected(x,z), Edge(z,y).
       Connected(x,y) :- Connected(y,x).
Component(x,min(y)) :- Connected(x,y).
```

The semantics of stratified aggregation follows the same logic as with stratified negation: each stratum of the program is executed one-by-one, while taking as input the result of the previous stratum. Most of the systems (see Table 2.1) support stratified aggregation.

Recursive Aggregation. Unfortunately, Datalog with only stratified aggregation is not sufficiently expressive to solve a large class of recursive problems, such as shortest paths. Few Datalog systems support some form of recursive aggregation (SociaLite, RecStep, and BigDatalog). From a theoretical perspective, early foundational work (Ross and Sagiv, 1992) proposed to use *monotonic aggregation* functions to support aggregation inside recursion. Informally, an aggregate function is monotonic if adding more elements to the multi-set being operated upon can only increase (or decrease) the value of the aggregate (e.g, min, sum). However, in order to guarantee the existence of a unique minimal fixpoint, several complex conditions must hold. As we will see next, most systems avoid this complexity by incorporating recursive aggregation with different choices on syntax.

SociaLite supports recursive monotone aggregation for aggregate functions that are meet operators and form a semi-lattice. More precisely, an aggregation operator F must also be the least-upper bound of some semi-lattice. Then, the semantics of a rule are extended by taking a group-by over the non-lattice elements and computing F for the lattice elements in the particular group. That is,

$$\{(x_1, \ldots, x_n, F(\{(x_0) \mid R(x_1, \ldots, x_n, x_0) \in \mathbf{I}\})) \mid R(x_1, \ldots, x_n, x_0) \in \mathbf{I}\}\,.$$

Specifically, min, max are meet aggregate operators, but sum is not.

Example 2.4. As an example, we show how to compute the length of the shortest path from some source node s to some sink node t in a graph represented through the binary relation Edge.

$$\begin{aligned}
\mathsf{Path(i,\ min(d))} &\mathrel{:\!-} \mathsf{Edge(s,\ i,\ d)}.\\
\mathsf{Path(i,\ min(d))} &\mathrel{:\!-} \mathsf{Path(i',\ d')}, \mathsf{Edge(i',\ i,\ d'')}, \mathsf{d} = \mathsf{d'} + \mathsf{d''}.\\
\mathsf{MinDistance(min(d))} &\mathrel{:\!-} \mathsf{Path(t,\ d)}.
\end{aligned}$$

At every iteration the body of the rule computes all the distances of paths starting at node s. The distances are then grouped by the end node, and then the min aggregate is applied to only keep the minimum distance. The final rule takes another minimum over all distances of paths that end at t.

Observe that SociaLite expresses aggregation in the head of the rule, in precisely one variable. Other languages like LogiQL in LogicBlox make different syntactic choices on how to represent aggregation.

RecStep and Myria allow aggregation inside recursion, but they leave it up to the programmer to make sure that the program terminates. Generally, it is not easy to check whether the program reaches a fixpoint after a finite number of iterations, especially as the program becomes more complex.

Flix (Madsen *et al.*, 2016), a Datalog system designed for static program analyses, also supports aggregation through monotone operations over lattices. In Flix, every predicate symbol is associated with a lattice. The partial order of the elements in the lattice induces a partial order on models (outputs) of the Datalog program. This allows to lift set-based positive Datalog to lattice-based positive Datalog, where all the nice Datalog properties and optimizations carry over. Using this formulation, Flix can support monotone aggregation operators inside recursion with well-defined semantics. Unfortunately, Flix does not have an efficient implementation yet. Thus, although operating directly on lattices is theoretically elegant, it is unclear whether it can be practical with large-scale data.

BigDatalog chooses yet another direction on how to support recursive aggregation (Mazuran *et al.*, 2013; Shkapsky *et al.*, 2015; Zaniolo *et al.*, 2017). Specifically, the standard aggregate operators (min, max, sum, count) are replaced by specialized versions named mmin, mmax, msum, mcount respectively. To demonstrate how these operators work, consider the following recursive rule taken from a program that computes shortest paths in a graph:

$$\mathsf{Path(x,y,\ mmin(d))} \text{ :- } \mathsf{Path(x,\ z,\ d'), Edge(z,\ y\ d''), d = d' + d''}.$$

Using this rule, a new tuple Path(x,y,d) will be added if either the endpoints are new, or the distance between x,y is smaller than any currently known length in the IDB. In this way, the sequence of values produced by the aggregate operator for a specific pair x,y will be monotone (decreasing). Contrast this to the use of the standard min operator, which adds any length that is not yet discovered. In other words, the idea behind the semantics of recursive aggregation in BigDatalog is to explicitly make the operator monotonic.

Finally, we should mention that Bloom^L, an extension of the language Bloom (Conway *et al.*, 2012), also extends Datalog with support for lattices and monotone functions. However, in Bloom^L the goal is to guarantee that the computation is *confluent*, meaning that a distributed computation terminates correctly with the same result regardless of the order in which messages are received over the network.

In summary, there is no widely accepted standard on how recursive aggregation can be supported in a practical system. The key underlying idea is to impose some form of monotonicity on the aggregate function. One design point is to use naive evaluation and burden the user with checking for termination. Another design point is to syntactically restrict its use, with the trade-off of losing some expressive power.

2.4 Other Language Extensions

In this section, we briefly mention other ways in which modern systems extend the (positive) Datalog language, in addition to negation and aggregation. As these extensions break the domain semantics, and so the earlier mentioned polynomial time complexity is not applicable here.

Arithmetic Operators. Many practical applications need some form of arithmetic calculations (e.g., addition, multiplication). For instance, we need addition to express the shortest path Datalog program mentioned in the above section. Such calculations can be easily incorporated in the syntax by viewing each arithmetic operator as a relation. For example, the assignment d = d'+d" is equivalent to the ternary predicate +(d,d',d"). However, because the relations corresponding to these arithmetic operators can be of infinite size, in general there is no guarantee that the fixpoint computation will terminate. There is a long line of research on how to guarantee well-behaved semantics in this case (Kifer, 1998; Ramakrishnan *et al.*, 1987), but systems generally leave this burden to the user.

As an example, consider the following Datalog program that runs the PageRank algorithm (Brin and Page, 1998), which combines aggregation with addition and division.

```
Rank(i+1,v,sum(r)) :- Nodes(v), r = 0.2/N.
                   :- Rank(i,u,s), Edge(u,v), EdgeCnt(u,c), r = 0.8 s/c.
```

The above program does not terminate, since the term $i + 1$ in the head of the rule will produce an infinitely increasing sequence of integers in the first attribute of Rank. Hence, there are no well-defined semantics here and it is up to the user to specify a termination condition for evaluation.

Functions. The addition of functions is also necessary for better expressivity (e.g., string transformations, list concatenation). Such functions require that the Datalog program does value invention (Cabibbo, 1995). As with arithmetic operators, under the presence of value invention many of the nice properties of Datalog programs, including guaranteed finiteness, are lost and become a responsibility of the programmer. Some systems (e.g., Soufflé) also support user-defined functions.

Other Features. Modern Datalog systems support other language features as well. Soufflé, SociaLite, and RecStep support primitive and complex types for the attributes in a relation (e.g., integers, strings, floats, etc). LogicBlox supports refmode predicates that associate with an entity a unique primitive identifying value. A few systems (SociaLite, Soufflé, BUD) support e the greedy choice operator, an operator that chooses a tuple non-deterministically in its evaluation (Greco *et al.*, 1992). The Vadalog System (Bellomarini *et al.*, 2018) supports existentially quantified rule heads, which is a central feature in the Datalog+/- family (Calì *et al.*, 2010) of Datalog-based languages for knowledge graph reasoning and ontology querying.

2.5 Distributed Datalog

At the end of this section, we present extensions of Datalog that allow to explicitly encode forms of distributed computation. We should emphasize that this should not be confused with the parallel execution of Datalog, where data distribution and exchange is decided by the system and not explicitly captured in the program.

NDlog (Network Datalog) is a language for declarative network specification (Loo *et al.*, 2006). It is a restricted variant of Datalog that allows for explicit control on data placement and movement, and it is intended to be computed in distributed fashion on a physical network. Such a network may not be fully connected, i.e., one node may not be able to directly communicate with all other nodes. In order to capture this restriction, NDlog uses addresses to specify the location of data on the network, and forces the computation to only work across links that are provided as input to the program and correspond to the actual physical links. We should note here that this restriction on communication is fundamentally different from parallel execution of Datalog, where it is typically assumed that all nodes can freely talk with each other.

WebdamLog (Moffitt *et al.*, 2015) is a version of distributed Datalog that captures applications where peers exchange messages and rules. In WebdamLog (Abiteboul *et al.*, 2011), a fact $R(t_1, \dots t_m)$ is also parametrized by a peer p as $R@p(t_1, \dots t_m)$. Informally, this means that the fact is located at p. A rule can then specify how data and rules are exchanged between the peers during evaluation of the program. For instance, consider the following rule, where the $ symbol is used to denote a variable:

```
album@joe($pid,$photo,$f) :- friend@joe($f), album@$f($pid,$photo).
```

This rule says that any friends of Joe should send their photos to the album that Joe has.

Dedalus (Alvaro *et al.*, 2010), a Datalog variant used for reasoning about distributed systems, parametrizes each fact not only with location but also time (modeled as consecutive integers). For example, $R(t_1, \dots, t_m)[T]$ means that fact $R(t_1, \dots, t_m)$ is true at time T. Dedalus supports deductive rules, which are derivations referring to the same time, and inductive rules, which are temporal rules that derive facts using facts from the previous time. Concretely, consider the following program:

```
R(x)[T] :- S(x,y)[T].
R(x)[T+1] :- S(x,y)[T], U(y)[T].
```

Here, the first rule is deductive, while the second rule is inductive. The inclusion of time in the syntax makes it possible to have clean minimal model semantics, even when expressing non-monotonic operations such as deletions. In addition to inductive and deductive rules, Dedalus also supports asynchronous rules, where the relationship between the time in the head and the body is unknown.

In all the above cases, the language itself implicitly specifies how the computation must proceed and how facts are communicated to the different locations. In particular, if a fact is in location @p and is needed in location @r to fire a local rule, then it must be sent from @r to @p. As we will see in the next section on parallel evaluation, Datalog engines hide this specification from the programmer and instead let the optimizer make the choice on how data should be shuffled around.

3

Evaluation

In this section, we discuss the evaluation strategies used by modern Datalog engines. We focus both on single-threaded implementations, as well as multi-threaded and parallel execution. Table 3.1 provides an overview of some of the properties of these engines with respect to their evaluation strategies, as well as the data layouts they use.

3.1 Semi-naïve Evaluation

Most modern Datalog engines, including BigDatalog, Soufflé, SociaLite, and RecStep, use semi-naïve evaluation as the core evaluation method. LogicBlox also used semi-naïve evaluation in its earlier versions, but

Table 3.1: Table of properties of Datalog Systems

	execution	**data layout**	**evaluation strategy**	**sync/async**
BigDatalog	parallel	row-store	semi-naïve	synchronous
Soufflé	multi-threaded	tries/B-trees	semi-naïve	asynchronous
SociaLite	multi-threaded/parallel	tail-nested tables	semi-naïve	asynchronous
RecStep	multi-threaded	row-store	semi-naïve	synchronous
LogicBlox	multi-threaded	B-trees/B$^\epsilon$-trees	semi-naïve	N/A
bddbddb	single-threaded	BDDs	semi-naïve	N/A
Myria	parallel	row-store	semi-naïve	both

then switched to other methods based on incremental view maintenance. Semi-naïve evaluation is a *bottom-up* technique. It improves upon naïve evaluation by taking care to use only newly discovered facts in subsequent iterations. Specifically, for every iteration $k \geq 1$ and IDB predicate R, it calculates the new facts $\Delta R_k = R_k - R_{k-1}$. Then, it uses ΔR_k to do an *incremental* computation of the next iteration R_{k+1} in the Datalog program through an incremental immediate consequence operator $\Delta \mathbf{T}_P$ (Algorithm 2). As an example, consider the following Datalog program that computes the transitive closure of a graph:

```
T(x,y) :- Edge(x,y).
T(x,y) :- T(x,z), Edge(z,y).
```

Then, $\Delta T_k = \pi_{x,y}(\Delta T_{k-1}(x,z) \bowtie Edge(z,y)) - \Delta T_{k-1}$. The above program is a linear program, so the incremental operator $\Delta \mathbf{T}_P$ only needs to consider the newly generated facts $\Delta \mathbf{I}_{k-1}$ to compute $\mathbf{I}_k$. For non-linear programs the incremental operator $\Delta \mathbf{T}_P$ will also depend on previous facts that have been discovered in earlier iterations, $\mathbf{I}_{k-2}$.

Algorithm 2: Semi-Naive Evaluation of Datalog

```
  Input: $\mathbf{I}$, Datalog program $P$
  Output: $P(\mathbf{I})$
1 $\mathbf{I}_0 \leftarrow \emptyset$, $\mathbf{I}_1 \leftarrow \mathbf{I}$ ;
2 $k \leftarrow 1$;
3 repeat
4     $k \leftarrow k + 1$;
5     $\Delta\mathbf{I}_k \leftarrow \Delta\mathbf{T}_P(\mathbf{I}_{k-2}, \Delta\mathbf{I}_{k-1}) - \mathbf{I}_{k-1}$;
6     $\mathbf{I}_k \leftarrow \mathbf{I}_{k-1} \cup \Delta\mathbf{I}_k$;
7 until $\Delta\mathbf{I}_k = \emptyset$;
8 return $\mathbf{I}_k$
```

Semi-naïve computation minimizes both computation, since facts are not repeatedly discovered across iterations, and communication in a parallel setting since only ΔR needs to be communicated across machines at every iteration. On the other hand, an efficient implementation of semi-naïve evaluation requires the construction of multiple indices,

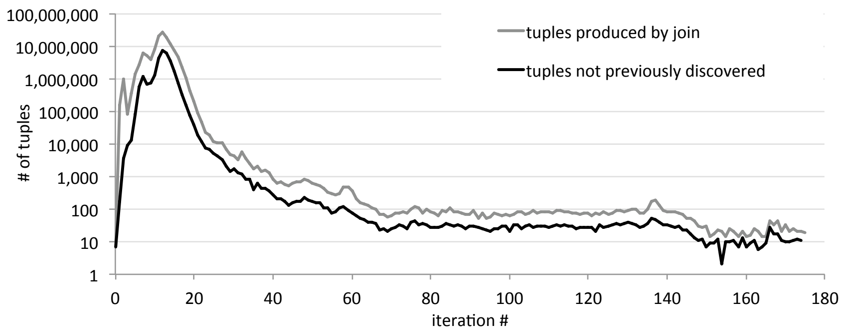

Figure 3.1: A plot of the number of facts generated per iteration for the Datalog program computing reachability.

since we potentially have to build a different set of indices for each Δ rule. Hence, while in theory semi-naïve evaluation is always more efficient, in practice there is no guarantee that it will run faster than naïve evaluation. The effect of index construction on evaluation will be discussed in more depth in Section 4.

Semi-naïve evaluation is particularly effective for *linear* Datalog programs. The gains are particularly visible in later iterations of a program, when the number of new facts can be very small (Shaw *et al.*, 2012) and hence the savings in communication and computation can help substantially reduce the total running time. For instance, Figure 3.1 shows how the number of new facts produced at later iterations of a Datalog program computing reachability are orders-of-magnitude fewer compared to earlier iterations.

In the end of this section, we will briefly discuss other evaluation methods, but for the subsequent sections we will focus on how different engines implement semi-naïve evaluation. Looking again at Algorithm 2, we observe that all of the computation happens at line 5, and it can be split into two fundamental operations:

Rule Evaluation: The first operation computes the incremental operator $\Delta\mathbf{T}_P(\mathbf{I}_{k-2}, \Delta\mathbf{I}_{k-1})$. Computing this operator amounts to executing a query q in Relational Algebra, where q is a (distinct) union of join-project queries. As we will see next, different sys-

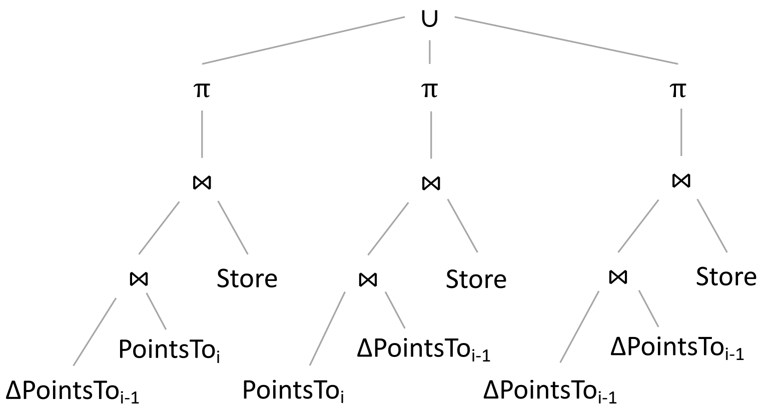

Figure 3.2: The query plan for evaluating a non-linear rule at iteration i.

tems perform this computation using different techniques and optimizations.

Difference: The second operation filters out the facts that we have already seen in previous iterations by computing the difference with $\mathbf{I}_{k-1}$. In this case, there are again different strategies on how to perform this computation.

3.1.1 Query Plans for Rule Evaluation

Within the general framework of semi-naïve evaluation, there are different methods of how each rule is executed. For example, consider the following rule from the Datalog program for points-to analysis:

$$\mathsf{PointsTo(x,y) :- Store(z,w), PointsTo(z,x), PointsTo(w,y).}$$

Computing this rule amounts to a query q in Relational Algebra (see Figure 3.2) which is a union of three join-project queries. (Recall that we also need to take another union across all rules with the same IDB predicate.) This query remains the same across all iterations, but the input to the query changes since the IDB relations are updated at every iteration.

Static vs Dynamic Plans. There are two design choices at this point, by choosing a static or dynamic query plan for q.

Static: Pick a fixed plan for q that remains the same independent of the iteration number (e.g., BigDatalog).

Dynamic: Attempt to find the best plan for q based on the most recent IDB inputs (e.g., RecStep).

The first choice may lead to suboptimal behavior when the optimal query plan for q varies across different iterations. For example, in some Datalog programs, the size of ΔR produced in the first few iterations might be much larger than a joining EDB table. Hence, if the query plan performs a hash join between these two tables, the hash table should be preferably built on the EDB side. However, as the ΔR produced in later iterations tend to become smaller, the build side for the hash table should be switched after some point.

The second choice of searching for an optimal query plan at every iteration requires that we collect data statistics across iterations, which may be a costly endeavor. RecStep mitigates this issue by collecting only lightweight statistics (e.g., hash-table size, size of tables) specifically tuned for the operators (join, projection) we want to execute. One thing to note here is that dynamic plans are more challenging to implement for systems that compile Datalog, such as Soufflé or SociaLite, since runtime information needs to be incorporated into plan generation.

Plan Optimization. Another design point is the choice of how to compute the best (static or dynamic) plan for q. RecStep translates rules with the same head to a single SQL query, which is then issued to the query optimizer of the underlying RDBMS, QuickStep, that finds the best plan for the current iteration. In the Datalog implementation on Myria and Naiad, the user must provide the plan manually to the system. BigDatalog translates each rule to a sequence of joins following a left-to-right-order, hence doing almost no search to find an optimal plan. Soufflé uses a fixed plan, but instead of considering multiple physical operators for joins (e.g., hash-join, sort-merge join), it limits the choice to a nested loop join for every rule. The nested loop join is accelerated by the use of specialized concurrent data structures such as B-trees and tries (we will discuss more on this on Section 4).

It is worth noting that plan optimization is even more challenging compared to optimizing SQL queries, since data statistics for later iterations do not exist and are very hard to estimate accurately. This is the reason why most systems opt for simple plans.

3.1.2 Difference Computation

In semi-naïve evaluation, the engine must compute the set difference between the newly produced facts and the entire IDB relation (R_{k-1}) at the end of every iteration, to generate the new ΔR_k. Since set difference is executed at every iteration for every IDB, it forms a computational bottleneck that must be highly optimized.

We mention next some of the data structures that have been used to speed up this bottleneck. In Soufflé, set difference is computed by checking membership against the data structure (B-tree) that holds the IDB relation: this data structure is designed to make this check efficiently. RecStep in contrast implements a specialized dynamic set difference operator to perform this task efficiently. This operator changes its core algorithm depending on the relative size of the IDB to the newly generated facts. Another option for computing the difference is to use versioned B-trees; it is relatively efficient to compute the difference of such trees since they will have a common history (i.e., the previous iterations).

3.2 Parallel Evaluation

In this section, we discuss how Datalog evaluation is parallelized by state-of-the-art systems. This capability is increasingly important as the volume of data that needs to be processed increases. We first present the theoretical underpinnings of parallelizing Datalog. Then, we discuss parallel strategies for both multi-threaded and distributed systems.

3.2.1 Theoretical Foundations of Parallel Datalog

Evaluating queries in Relational Algebra (the core fragment of SQL) is in the complexity class AC_0, which corresponds to boolean circuits with polynomial size and constant depth. Since the depth of a circuit is a proxy of how well computation can be parallelized, this means that for all practical purposes RA is embarrassingly parallel. Note that RA roughly corresponds to Datalog with negation but without recursion. On the other hand, Datalog evaluation is P-complete (Abiteboul *et al.*, 1995). Since it is widely believed that the P-complete class includes inherently sequential problems, it is highly unlikely – at least theoretically – that every Datalog program can be significantly sped up using parallelism.

Evaluating linear Datalog programs, such as transitive closure or reachability in graphs, belongs in the complexity class NC (Kanellakis, 1986; Cosmadakis and Kanellakis, 1986). More generally, if a Datalog program has the *polynomial fringe property*, which says that every fact in the output has a proof tree of polynomial size, evaluation is in NC (Ullman and Gelder, 1988). Every piecewise linear Datalog program (a slight generalization of linear programs) has the polynomial fringe property and is thus in NC. At a high level, NC contains the problems that can be solved by boolean circuits of polynomial size but logarithmic depth. Hence, from a theoretical perspective such Datalog programs admit an efficient parallelization. In practice, one can think of such programs as programs that terminate – after some rewriting – within a logarithmic number of iterations. We should note here two things: (*i*) most Datalog programs observed in graph processing and business analytics are linear, and (*ii*) linear programs are the ones where semi-naïve evaluation is particularly efficient.

On the other hand, non-linear programs (such as most points-to program analyses) are generally not in NC. Since it is widely believed that NC is strictly contained in P, we do not expect such programs to be easily parallelizable. For instance, the simple Datalog program below is P-complete:

```
T(x) :- U(x).
T(x) :- T(y), T(z), R(x,y,z).
```

Unfortunately, there exists no general method to decide whether a Datalog program is in NC or not, with the exception of a result on a small fragment of Datalog programs called chain queries (Afrati and Papadimitriou, 1987).

3.2.2 Shared-nothing Datalog Engines

Shared-nothing architectures are typically deployed to scale out data processing (Stonebraker, 1985; DeWitt and Gray, 1992). In a shared-nothing architecture, a large number of independent processors/machines are connected via a fast communication network. Several big data systems have been deployed on top of a shared-nothing architecture (Zaharia *et al.*, 2016; Malewicz *et al.*, 2010; Dean and Ghemawat, 2004).

Shared-nothing architectures have been the core of several scalable Datalog systems as well. BigDatalog provides scalable evaluation of Datalog on top of Spark (Shkapsky *et al.*, 2016). In Wang *et al.* (2015), the authors consider asynchronous Datalog evaluation on top of Myria. SociaLite also has a shared-nothing parallel implementation (Seo *et al.*, 2013b). Finally, Datafrog is a lightweight implementation of Datalog using the underlying Naiad system (Murray *et al.*, 2013).

Since semi-naïve evaluation can be thought of as a sequence of relational queries, parallel Datalog engines borrow many techniques from traditional parallel query processing (Kossmann, 2000). Both IDB and EDB relations are partitioned (sharded) to the machines using hash-partitioning or another partitioning method. Using the partitioning scheme, the engine can then pick a parallel query plan for every rule, and also form a communication pattern of where to send the new facts that are produced at each machine. However, the decision on when to send the data and whether (and when) synchronization is needed differs across the engines. Over the next part, we will discuss these design choices in detail.

Data Partitioning Strategies. The partitioning strategy is a critical design choice for efficient parallel evaluation in Datalog. Although the partitioning of an EDB relation is done once at the beginning, IDB relations may have to be repartitioned at every iteration as new facts are produced. This happens because a parallel join operator needs to partition each relation by their join key, but the newly generated tuples may end up in a different block of the partition. Hence, in order to minimize data communication – and thus the total running time – it is important to choose an appropriate partitioning strategy.

A rich line of research on partitioning strategies for Datalog exists since the 1990s (Ganguly *et al.*, 1990; 1992; Wolfson, 1989; 1990; 1993; Zhang *et al.*, 1995). These partitioning strategies are very general, in the sense that they allow for adding extra conditions to the bodies of rules in the form of restricting predicates. For example, consider a hash function h that maps values from the domain to one of the p machines. Then, add the following predicates @h to each rule of the linear transitive closure program:

```
T(x,y) :- Edge(x,y) @h(x).
T(x,y) :- T(x,z), Edge(z,y) @h(z).
```

We interpret this as follows for the second rule: for any two facts that join on $z = a$, their result must be produced at machine $h(a)$. This restriction implicitly describes a parallel evaluation strategy: relation Edge is partitioned on its first attribute (just once), and any new fact $T(a, b)$ discovered at iteration i must be sent to machine $h(b)$ for the next iteration. Then, the rules can be locally executed to produce the new facts. Note that this is equivalent to a parallel hash join followed by a reshuffling step, but one can express much more complex strategies using more general predicates. It is worth contrasting this way of representing a partitioning strategy (hidden from the programmer) with the location specifiers used in distributed Datalog (Dedalus, WebdamLog).

Existing parallel Datalog engines deploy simpler partitioning methods. The standard partitioning method for parallel execution in Datalog is *hash partitioning* (see BigDatalog, SociaLite, Myria). Hash partitioning is a technique widely used in parallel RDBMSs. Every relation (extensional or intentional) chooses a subset of its attributes to be partitioned by. Since there can be a mismatch between an operator's input partitioning and the chosen partitioning, a *shuffling step* is often required to repartition the relation. For instance, in the example above,

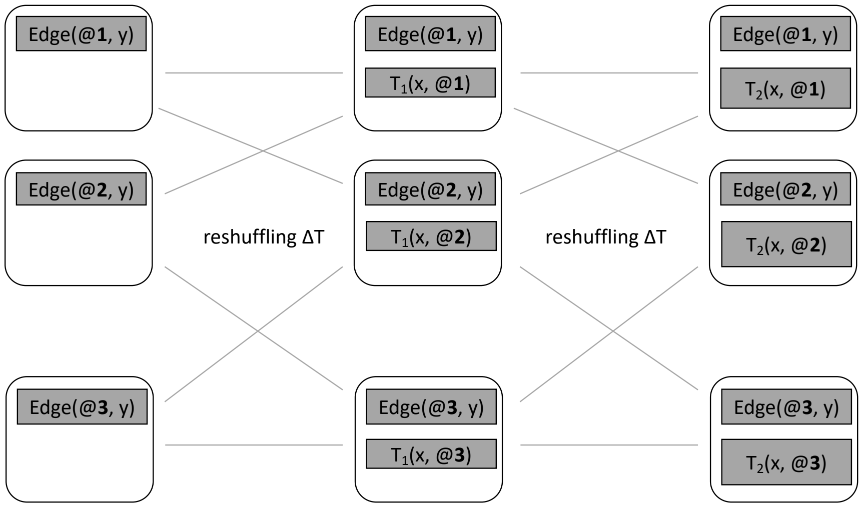

Figure 3.3: Communication pattern and partitioning for linear transitive closure. The newly produced facts at every machine must be reshuffled at the end of every iteration using the second attribute as partitioning key.

the newly produced facts of T will not be partitioned at all, so we need to partition them using the second attribute (see Figure 3.3).

BigDatalog chooses by default the first attribute of a relation as its partitioning key, while SociaLite leaves this choice to the user. SociaLite also employs a different partitioning method, *range partitioning*. Range partitioning is implemented by dividing the range into consecutive sub-ranges that are evenly distributed across the machines. In SociaLite, the users themselves have to specify the communication pattern according to the partitioning method. This may work for small programs, but is infeasible for larger programs with more rules.

Finally, we should mention that in cases where an EDB relation is relatively small, the system could also consider *broadcasting* the relation instead of using partitioning. Broadcasting is particularly effective in Datalog, since it needs to be done once in the first iteration, but then it requires no communication for the subsequent iterations.

Decomposability. Some Datalog programs allow for a particularly efficient type of parallel evaluation, where after the initial partitioning of the EDB relations there is no data transfer of IDB facts between machines and no redundancy of computation as well (the second condition is needed because one could replicate a sequential computation across all machines). A program that admits such an execution is called *decomposable*. It will be helpful to explain decomposable programs using the following example, which computes all pairs of nodes in a graph connected by an odd-length path:

```
T(x,y) :- Edge(x,y).
T(x,y) :- T(x,z), Edge(z,w), Edge(w,y).
```

For this program, we deploy the following strategy: we hash-partition T according to its first attribute, and broadcast the EDB relation Edge to all machines. The advantage of this strategy is that every new fact generated by the recursive rule will automatically be in the correct block of the partition, and since Edge is broadcast, there is no need for any repartitioning. Figure 3.4 depicts this strategy. This strategy does not produce any extra work, since each fact in T will be produced by exactly one machine. Additionally, since there is no need for communication (with the exception of the initial partitioning step), each machine can do its computation without the need of any further synchronization. On the other hand, this strategy may be infeasible if the EDB relations are so large that they can not be stored in a single machine.

Unfortunately, there is no systematic way to decide whether a Datalog program is decomposable or not (Wolfson and Ozeri, 1990). However, there exist a number of sufficient conditions that imply decomposability based on the existence of a *pivot*, a property identified first in Wolfson and Silberschatz (1988) in the context of single-rule programs. In our example, the pivot consists of the first attribute position of T. The notion of a pivot was later generalized to full Datalog as *generalized pivoting* (Seib and Lausen, 1991).

The generalized pivoting technique has been adopted in BigDatalog and is part of its compiler. If the compiler can find a generalized pivot set, the Datalog program is decomposable and an appropriate partitioning is constructed based using its pivot set.

Recently, a line of work has looked at how decomposable strategies can be generalized to incorporate more general partitioning meth-

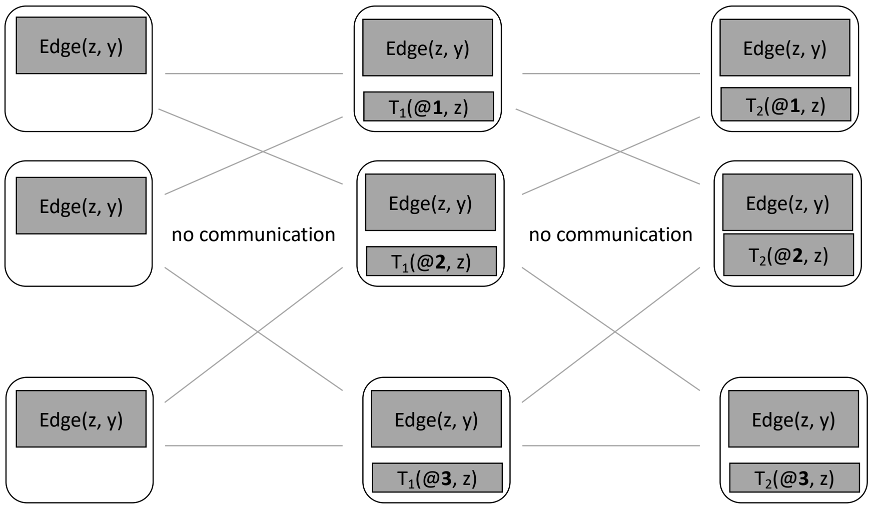

Figure 3.4: Communication pattern and data partitioning using a decomposable strategy. There is no communication or synchronization necessary.

ods (Ketsman *et al.*, 2020), specifically using the Hypercube technique (Afrati and Ullman, 2010; Beame *et al.*, 2017).

Synchronous vs Asynchronous Computation. An important design point in parallel computation of Datalog is whether the execution should be synchronous or asynchronous. In a synchronous execution model, at the end of every iteration there exists a synchronization barrier, that is, all machines need to make sure they have completed their computation before progressing to the next iteration. This execution model is a natural fit to semi-naïve evaluation, which splits computation in iterations and uses only the recently discovered facts (in the previous iteration) to produce new facts. However, synchronous execution introduces a possible computational bottleneck, since a straggler machine or the existence of data skew will make the computation slower. This problem is exacerbated in Datalog (compared to non-recursive computation), since the number of synchronization points depends on the number of iterations and can be very large.

On the other hand, asynchronous processing has the benefit of resilience against uneven load distribution because each machine performs computation without the need of synchronization. However, there are several drawbacks to the use of asynchronous execution methods. First, not all Datalog programs are amenable to asynchronous evaluation. For example, the existence of negation in the rules often requires that the execution method uses some synchronization at this point, otherwise the output result is not guaranteed to be correct. Second, not all parallel systems are amenable to asynchronous execution of their operators, since this requires that the data can be pipelined from one operator to the next without synchronization. For example, Spark does not natively support a pipelined execution of its data shuffling step, and thus it needs to be extended before it can support asynchronous execution. Third, asynchronous evaluation can lead to a larger computational load, and in particular computation that is replicated across different machines. The reason for this redundancy is that it is possible that a machine that is further ahead in terms of computation accesses stale data in some other machine instead of waiting for the newer data to arrive.

BigDatalog adopts the synchronous execution model, with a synchronization barrier at the end of every iteration. SociaLite chooses a mixture of synchronous and asynchronous execution. In particular, synchronization is only needed when computation moves from one stratum to the next stratum of the program; within the same stratum, results are communicated as soon as they are ready with message passing (with the exception of possible batching).

In Loo *et al.* (2006), the authors implement NDlog by using an asynchronous distributed variation of semi-naïve evaluation called *pipelined semi-naïve evaluation (PSN)*. In PSN, a new tuple is processed on a node as soon as it is received, without waiting for the current iteration to complete. In order to guarantee that the same inference will not be computed multiple times, PSN assigns to each new tuple a timestamp (instead of a number based on the iteration it was generated), and the new tuple can be matched only with tuples that have the same or an older timestamp. PSN thus allows for a fully pipelined evaluation, with the trade-off that set-oriented processing is not fully utilized.

In Wang *et al.* (2015), the authors implement and compare the synchronous and asynchronous mode of execution over the same parallel system, Myria. Their study shows that none of the two execution models is superior. For instance, asynchronous computation was much faster for the task of computing the connected components of a graph, while synchronous computation performed considerably better for the task computing the least common ancestor for pairs of publications in a citations graph. The choice of which strategy is better depends on the number of iterations of the program and the number of intermediate facts produced during execution. Since both of these parameters are only known at runtime and are difficult to estimate, this optimization choice becomes a challenging problem in practice.

Asynchronous and Coordination-Free Programs Extreme examples of asynchronous Datalog computations are found in the context of declarative networking (Loo *et al.*, 2009), where Datalog and extensions of Datalog are used to program networks with minimal synchronization and blocking mechanisms in place. In such settings, a great deal of effort goes into understanding the distributed semantics of programs.

Example 3.1. Take as an example the single-rule semi-positive Datalog program,

$$\mathsf{Abort}() \mathrel{:\!-} \mathsf{not}\ \mathsf{VotesForCommit}(\mathsf{ip}), \mathsf{Machine}(\mathsf{ip}).$$

that instructs machines to abort a transaction if one of machines in the network fails to vote for a commit. Timing matters for this program, as the intended semantics is only obtained if the rule executes *after* all machines have had enough time to populate the VotesForCommit relation with their vote.

This example is in stark contrast to the case for programs written in positive Datalog, whose semantics preserves without any artificial blocking mechanisms in place. The outcome of positive Datalog programs eventually converge to agree with the program's non-distributed sequential semantics, despite temporal non-determinism caused by disorderly firing rules, network delays, and re-applying rules over inflating instances.

Based on the observed correlation between negation and need for blocking, Hellerstein (2010) raised the question to what extent blocking and synchronization are required by the semantics of Datalog programs; conjecturing a strong relationship between the (eventual) consistency of program executions and whether the program is monotone. A formal exploration of CALM – the common acronym to refer to this relationship – was done first by Ameloot and Van Den Bussche (Ameloot *et al.*, 2013), who, for a particular definition of non-blocking (called *coordination-freedom*) showed that all, and only, monotone queries (not necessarily written in Datalog) have a non-blocking strategy. Their model assumes only weak forms of network reliability (like guaranteed at-least once arrival of messages) and puts no restrictions on the way data is initially partitioned and placed on machines in the network. In later work, it was shown that there is a leverage between the semantics of programs that allow non-blocking computations and system knowledge about the initial data partitioning (Ameloot *et al.*, 2016). One such result is that semi-monotone programs require no blocking if a partitioning-strategy is used and known by the machines in the system. (In Datalog terms, these contain the semi-positive programs). The crux underlying this result is that a semi-monotone query can be seen as a monotone query taking as input over the input relations *and* their complements (Abiteboul *et al.*, 1995).

In a parallel setting, a partitioning strategy makes individual nodes responsible for certain facts, these nodes can thus infer that "facts that are absent on a node, but fall under that node's responsibility, cannot reside on other nodes either". Through distributed effort, the entire complement relations become computable without blocking and therefore allow to also compute the remaining (now monotone) computation without blocking.

Example 3.2. For an example, consider the semi-positive Datalog program,

$$\mathsf{Poi(x)} :\!\!-\ \mathsf{Supplier(x), not\ Customer(x).}$$

asking for all suppliers who are not in the customer relation. Let us assume that the customer relation is range-based (or hash-based) partitioned over a number of machines, and that all these machines have a local predicate $\mathsf{Customer}_{\text{local}}$ for the fragments of customers at hand and a local predicate $\mathsf{Range}_{\text{local}}$ representing the range that the particular machine is responsible for. Then, every machine can execute a rule collecting suppliers that are not in the customer relation at hand but within the partition's range,

$$\mathsf{NoCustomer(x)} :\text{-}\ \mathsf{Supplier(x)}, \mathsf{not}\ \mathsf{Customer}_{\text{local}}\mathsf{(x)}, \mathsf{Range}_{\text{local}}\mathsf{(x)}.$$

which, through distributed effort, computes the complement of the customer relation. The positive Datalog rule,

$$\mathsf{Poi(x)} :\text{-}\ \mathsf{Supplier(x)}, \mathsf{noCustomer(x)}.$$

completes the execution.

The theoretical upper bounds unfortunately yield no direct practical algorithms and rely on sending all facts to all nodes, which is the best one can do for arbitrary monotone queries (in no particular language). As we discussed in the previous section, systems like BUD (Alvaro *et al.*, 2011; Conway *et al.*, 2012) explore these upper-limits through extensions of Datalog with more expressive rules, which are still monotone, but over arbitrary user-defined lattices. Since some of the features, like arithmetics, do not guarantee convergence and termination, this requires some reasoning by the programmer.

3.2.3 Shared-Memory Datalog Engines

In addition to distributed environments, another line of work has looked at scaling up Datalog in shared-memory settings: RecStep (Fan *et al.*, 2019), Datalog-MC (Yang *et al.*, 2017), Soufflé, and Graspan (Wang *et al.*, 2017). In such an architecture, computation is spread across multiple units, but the main memory and disk are shared resources. The first three systems focus on main-memory settings, where the data fits in memory, while Graspan considers also disk movement. However, the expressivity of Graspan is restricted to binary relations (graphs) and hence it does not support the full positive Datalog.

While in a distributed setting the key concern is to minimize communication and synchronization, in a shared-memory architecture the core challenge is to handle the contention on the data structures that are repeatedly accessed at every iteration. Thus, there is a common thread across all shared-memory Datalog engines to construct specialized data structures and specialized operators with a high degree of concurrency, instead of using a generic concurrent implementation. In what follows, we dive deeper on how these engines work.

Datalog-MC hash-partitions tables and executes the partitions on cores of a shared-memory multicore system using a variant of hash-join. The decision on which attributes should be used for the partitioning (called *discriminating sets*) is made by an optimizer and takes into account the query workload and the estimated cost of the best plan for each query in the query workload. To evaluate Datalog in parallel, the rules are represented as *and-or* trees, where an "and" corresponds to a join operator, and an "or" corresponds to an operation like filter or union. During evaluation, the and-or tree is traversed in a bottom-up fashion, by sending new facts from the leaves of the tree towards its root. The parallelism comes from creating copies of the and-or tree, one for each partition, that can be evaluated simultaneously. This formulation allows Datalog-MC to reason in a fine-grained manner on whether a read or write lock is needed to move the evaluation forward, hence minimizing contention. Datalog-MC also uses B-trees as the default indices to speed up search.

RecStep takes the approach of using a very fast main-memory RDBMS as its underlying engine. It compiles Datalog directly to a sequence of SQL queries, and lets the RDBMS optimize and execute each query. All relational operators in QuickStep have high-performance parallel implementations, but special care needed to be taken for the deduplication operator. Recall that deduplication happens at every iteration for every idb relation, and it is necessary to avoid redundant computation. RecStep implements the deduplication operator by a global latch-free hash table, in which tuples from each data partition can be inserted in parallel. The size of the hash table is also tuned by the optimizer to use as many pre-allocated buckets as the available memory allows, thus preventing memory contention.

Soufflé achieves high scalability by parallelizing the execution of the nested loops in C++. Recall that Soufflé evaluates each rule by executing an (indexed) nested loop join over the atoms in the body of the rule. Parallelizing the execution of each loop means that when Soufflé has to check whether a fact exists in a relation when Soufflé inserts a new fact in the relation, the engine must be able to perform these operations concurrently. To solve this issue, Soufflé has built its own concurrent data structures specialized to Datalog execution (Jordan *et al.*, 2019a; 2019b). We will discuss these techniques in more detail in Section 4.

Graspan operates only on graphs (i.e., binary relations), and creates partitions of the edges by splitting according to the source vertex. In each iteration, two partitions are loaded into memory and joined to produce new facts – this can be done in parallel across all partitions. When a partition becomes too large, Graspan dynamically repartitions to smaller partitions. We should note that the focus of Graspan is to minimize memory-disk movement, which is not a focus of all other Datalog engines.

Finally, we should also point out here a line of work that uses GPUs to achieve high-performance parallel Datalog processing in a shared-memory setting (Martinez-Angeles *et al.*, 2013). Their basic data structure is an array of tuples which allows for duplicate facts. Thus, after every relational operation, explicit duplicate elimination is performed – which for some cases dominates execution time. The system also takes care to minimize the movement between GPU and CPU by keeping facts in the GPU as long as possible.

3.3 Compilation vs Interpretation

A different design point on how to architect a Datalog engine is whether the Datalog program is compiled or interpreted. This is a particularly interesting trade-off in the design space, since as we discussed Datalog execution is a dynamic process: the number of iterations is not known a priori, and it is possible that the optimal query plan in every iteration is different. Hence, compilation limits the amount of on-the-fly optimizations one can apply when computing a Datalog program.

Among the Datalog systems we are focusing on, Soufflé compiles down to a parallel C++ program. SociaLite also compiles to Java code (using an underlying distributed file system). On the other hand, RecStep produces a sequence of SQL statements that run on top of an RDBMS, while BigDatalog runs on top of (modified) Apache Spark.

3.4 Other Evaluation Methods

Apart from bottom-up evaluation strategies, there exists another family of evaluation methods that operates in top-down fashion. For example, in the query-subquery (QSQ) method (Abiteboul *et al.*, 1995), we start from the goal and attempt to find facts that match the goal. This approach can be beneficial when it is possible to obtain the output without computing the full model of the Datalog program. However, top-down approaches are unlikely to work once the input relations get really large as with many modern applications (e.g., program analysis, graph processing), and none of the systems presented in this monograph use them. We should note here that languages that include Datalog as a special case (e.g., Prolog), use top-down (or goal-oriented) evaluation as the default evaluation method.

4 Data Layouts and Indices

In this section, we discuss how data is represented in modern Datalog engines. First, we present the various data layouts used to facilitate Datalog evaluation. Even though most engines prefer the standard row-oriented representation, several engines use different data layouts to optimize for specific application domains. For example, to accelerate the evaluation of context-sensitive program analyses, bddbddb uses binary decision diagrams to efficiently represent a relation. Socialite uses a generalized adjacency list layout to represent graphs, while RecStep uses bit matrices to represent dense relations of small arity. In the second part, we discuss various indexing techniques used by modern Datalog engines.

4.1 Data Layouts for Datalog

Most Datalog engines use the standard *row-oriented* representation for both the IDB and EBD relations. We will use the instance in Table 4.1 of a ternary relation $R(A, B, C)$ as a running example throughout this section, depicted using a row-oriented representation.

A row-oriented representation also helps interfacing with any underlying Relational DBMS that the engine may use. We should note here that in most applications of Datalog the number of columns on each relation is quite small, and hence column-oriented representations have not been used in practice.

Even though row-oriented representations are prevalent, there are Datalog engines that use other representations targeted to optimize the performance of specific applications. Most of these data layouts exploit properties of the input distribution (e.g., density) to design representations that both save space and speed up the performance. We next present in detail these data layouts and discuss their target application.

Table 4.1: Instance of a ternary relation $R(A, B, C)$ used as a running example.

A	B	C
0	1	0
0	1	1
0	3	0
0	3	1
2	0	1
2	1	1

4.1.1 Binary Decision Diagrams

bddbddb uses Binary Decision Diagrams (BDDs) to perform pointer analysis on large programs. As we have seen, pointer analysis can be naturally expressed in Datalog. However, for certain types of pointer analysis, such as *context-sensitive analysis*, the IDB relations we want to compute grow exponentially large with the size of the code. Thus, even for relatively small EDBs it becomes inefficient to represent the IDB relations in a row-oriented store. BDDs offer a much more compact representation for these relations, hence allowing for an improved performance.

In order to represent a relation $R(A_1, \dots, A_d)$ as a BDD, we first need to perform a binary encoding of the relation. Assume that each value of attribute A_i comes from a domain D_i, which takes values from $\{0, 1, \dots\}$. We can then encode each value with at most $\log_2(n)$ bits

A		B		C
b_1	b_2	b_3	b_4	b_5
0	0	0	1	0
0	0	0	1	1
0	0	1	1	0
0	0	1	1	1
1	0	0	0	1
1	0	0	1	1

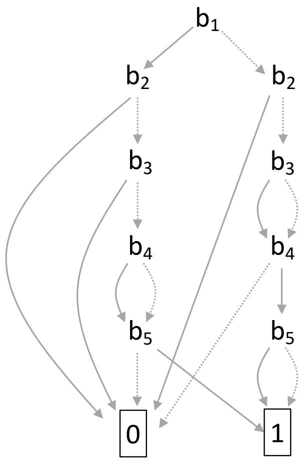

Figure 4.1: Binary encoding and corresponding OBDD for the running example. The solid edges correspond to 1-edges and the dashed edges to 0-edges.

where n is the size of the relation. For our running example, we have $D_A = \{0, 1, 2, 3\}$, $D_B = \{0, 1, 2, 3\}$ and $D_C = \{0, 1\}$. Thus, we need 2 bits for attributes A, B and 1 bit for attribute C, for a total of 5 bits. We can now think of an instance as a boolean function f that maps each element from $D_1 \times D_2 \times \cdots \times D_d$ to $\{0, 1\}$: $f(t) = 1$ if a tuple t exists in the instance, otherwise $f(t) = 0$. Figure 4.1 (left) depicts the encoding as a boolean function for the running example (only the entries with $f(t) = 1$ are shown).

A BDD is a directed acyclic graph (DAG) with a single root node and two terminal nodes: 1 and 0. Each non-terminal node is an input variable and has two outgoing edges: a 1-edge and a 0-edge. The 1-edge represents the case where the variable for the node is true, and the 0-edge represents the case where the variable is false. To evaluate a BDD for a specific input, one starts at the root node and, for each node, follows the 1-edge if the input variable is 1, and the 0-edge if the input variable is 0. The value of the terminal node that we reach is the value of the BDD for that input. Figure 4.1 (right) depicts a BDD for the running example using the binary encoding on the left.

bddbddb uses a variant of BDDs called Ordered BDDs, or OBDDs. An OBDD has the constraint that on all paths from the root to a terminal node the variables we meet follow a fixed variable order. The choice of the variable order can make a huge difference on how compact the BDD is, but unfortunately computing the optimal order is an NP-hard problem (Bollig and Wegener, 1996). bddbddb heuristically attempts to choose a good order. In addition, when we construct the BDD we take care to collapse common subgraphs into a single graph and share these nodes. This means that the more commonalities there are in the structure of the BDD, the more space-saving this representation will be. This is exactly the reason why BDDs are so space-efficient (and thus performant) when used for context-sensitive program analysis.

4.1.2 Tail-nested Tables

Tail-nested tables are a data layout explicitly designed for representing graphs. They have been used as the core data representation for Socialite (Seo *et al.*, 2013a), a Datalog engine geared towards social network analytics. Tail-nested tables generalize the notion of an adjacency list to represent edges in a graph.

Formally, the last column of a tail-nested table may contain pointers to two-dimensional tables, whose last columns can themselves expand into other tail-nested tables. The level of nesting is arbitrary and it is indicated by parentheses in the table declarations.

For example, a tail-nested representation of a ternary relation R with three attributes A, B, C can be written as $R(A, (B, C))$. In this case, the data layout has one column for attribute A with pointers to a two-dimensional table, where each row of the table stores a value of attribute B and a value of attribute C. To apply this to the graph setting, let attribute A denote the source of an edge, B its target, and C a property of either the edge or the target node. Figure 4.2 depicts the data layout for this example.

There are two benefits of tail-nested tables when representing graphs. First, the edges of the graph can be compactly stored, since edges with the same source need to store the source node once. Second, the data representation itself forms an index that can speed up the evaluation

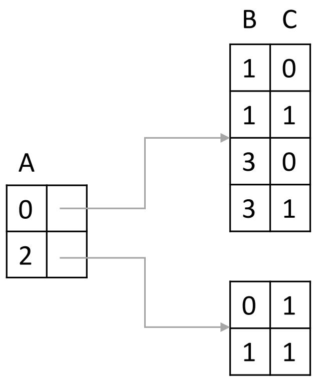

Figure 4.2: The tail-nested table $R(A, (B, C))$ for the running example.

of Datalog programs. Indeed, it is straightforward to see how both hash-joins and nested-loop joins can be modified to handle tail-nested tables as the input representation.

4.1.3 Tries

A data representation of similar flavor to tail-nested tables are *tries*. A trie is a tree data structure, where each level of the leaf corresponds to an attribute of the relation. Moreover, each tuple corresponds to a unique path from the root to a leaf of the trie. To speed up search, one can also choose to order the children of a node in the trie according to some fixed order that is consistent throughout the same level. We should note here that a trie requires a specific ordering of its attributes (as with an OBDD) and different orderings will lead to possible different running times for a program. Finding the optimal attribute ordering for a given workload is a hard optimization problem. Figure 4.3 shows the trie for the running example if the attribute order is A, B, C.

Tries offer compact representation similarly to tail-nested tables. Moreover, they are particularly effective when used to compute non-recursive Datalog programs through worst-case optimal algorithms – see for example the Leapfrog TrieJoin algorithm (Veldhuizen, 2014).

The LogicBlox engine uses tries as a virtual abstraction over more conventional data structures. Soufflé also implements a specialized

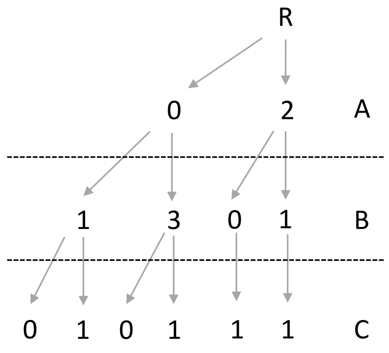

Figure 4.3: A trie for the running example when the attribute ordering is A, B, C.

version of a trie called *brie* (Jordan *et al.*, 2019b), which is a specialized concurrent data structure that can be viewed as a combination of a trie (meaning the tuples are encoded in the root-to-leaf path) and a B-tree. This data structure is particularly effective for relations of *high density*. Intuitively, density measures the fraction of points that in the relation out of all the possible points that can be constructed using the values in that relation. For example, a binary relation $R(A, B)$ has the maximum possible density if it is the Cartesian product of the two attribute columns. High density implies a lot of repetition, and hence compacting the representation is beneficial. As we will see next, bit matrices are also effective in representing high density relations.

4.1.4 Bit Matrices

In evaluating Datalog programs, it can happen that we start with sparse EDB relations with a relatively small active domain, but end up with large and dense output relations. This phenomenon has been observed in both graph analytics and program analysis (recall for example the discussion on the use of BDDs). A typical example of this behavior is the classic Datalog program of computing the *transitive closure* on a directed graph:

$$\mathsf{TC(x,y)} \text{ :- } \mathsf{Edge(x,y).}$$
$$\mathsf{TC(x,y)} \text{ :- } \mathsf{TC(x,z), Edge(z,y).}$$

A	B
1	2
1	4
2	1
2	5
3	4
5	5

B

A

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	1	0
4	0	0	0	0	0
5	0	0	0	0	1

Figure 4.4: A binary relation $R(A, B)$ along with its representation as a 2-dimensional bit matrix.

In the above program, even if the input graph is sparse (meaning that the number of edges is $O(n)$, where n is the number of nodes), the IDB TC can grow very large after only a few iterations and can become as large as $O(n^2)$. A row-oriented store will thus grow large and consequently fast join processing (e.g., using a hash join) will become memory-costly and slow. In the extreme, the intermediate table will become too big to fit in main memory, thus incurring additional I/O overhead.

To overcome this issue, RecStep proposes to use a specialized data structure, called a *bit-matrix* (Fan *et al.*, 2019). A bit-matrix representation is particularly effective when the active domain of the attributes in an IDB relation is relatively small, but the final result can grow large, hence ending up with a dense relation.

We describe here the bit matrix for binary relations, but the technique can be extended in a straightforward way to relations of higher arity. Let $R(A, B)$ be a binary IDB relation, with active domain $\{1, 2, \ldots, n\}$ for both attributes. Instead of representing R as a set of tuples, we represent it as an $n \times n$ 2-dimensional bit matrix M. If $R(a, b)$ is a tuple in the instance, the bit at the a-th row and b-th column, denoted $M[a, b]$ is set to 1, otherwise it is 0. Figure 4.4 shows an example of such a representation.

The reader should note here that a relation in bit-matrix format can be updated during each iteration by simply setting bits from 0 to 1 whenever a new fact is generated. Moreover, the space required to represent the IDB remains constant throughout the iterations. One additional important feature of the bit-map representation is that it allows easy parallelization in a multi-core setting, by partitioning the bit-matrix row-wise and assigning each partition to a different thread. In fact, this partitioning technique leads to an almost zero-coordination algorithm for every iteration.

4.2 Indexes

In addition to the choice of data layout, a second important consideration of how data is represented in the context of a Datalog engine is what indexes should be built. Since most Datalog engines in their core perform relational data processing (and in particular join processing), the use of standard index structures such as hash-tables and B-trees does speed up the evaluation and is common in many engines. Nevertheless, as we will see next, some engines (Soufflé, LogicBlox) choose to implement their own specialized indexes.

4.2.1 Specialized Indexes for Datalog

B-trees have been successfully used in multicore data processing (Graefe, 2010; Sewall *et al.*, 2011), and they have also been used to accelerate parallel Datalog evaluation in Soufflé. We should at this point examine closer Datalog with synchronous versus asynchronous evaluation. In the case of synchronous evaluation, there is a clear separation of reads and writes in the data structure: at every iteration read operations (evaluating the rules) are always followed by several write-only operations (inserting the new facts in the index). An asynchronous evaluation strategy does not have this property, since the order of reads and writes will be arbitrary. Soufflé takes advantage of this separation by building an in-memory B-tree data structure that has a new optimistic fine-grained locking scheme using optimistic read-write locks (Jordan *et al.*, 2019a). In addition to these algorithmic ideas, the actual implementation used in Soufflé carefully tunes several other features to improve performance for Datalog.

4.2.2 Index Selection

There is a large body of literature on index recommendation techniques for RDBMSs, most of them based on heuristic methods (Agrawal *et al.*, 2000). Unfortunately, these techniques are not sufficient when applied to evaluating Datalog programs. There are multiple reasons for this. First, indexes are needed for both IDB and EDB relations, and the former need a *dynamic index*, since the relation changes after every iteration. Second, some Datalog programs – especially in program analysis – consist of hundreds of relations and rules, which means that typical index recommendation techniques that do not scale fail.

Most Datalog engines require that the user manually specifies what indexes should be used (e.g. Doop, Socialite), or default to simple strategies. For example, RecStep does not build any indexes up front, but the underlying RDBMS will build on-the-fly hash indexes for the join and deduplication operator. For the engines that use indexes based on a fixed attribute order (e.g., tries in LogicBlox, BDDs in bddbddb), an additional challenge becomes the choice of the optimal attribute order. In general the latter problem is intractable (especially since different rules possibly have different optimal orders), hence the only solution is to use heuristic methods based on collected data statistics.

Souffle uses a different approach and instead automatically constructs an appropriate set of indices (Subotić *et al.*, 2018). Recall that Soufflé executes each rule as a nested loop join, which is sped up via the use of B-trees. The order of the predicates in the nested loop determines which B-trees would be useful to speed up the given rule. The access pattern for a B-tree in Datalog evaluation is typically a selection with equality predicates on a subset of the attributes. For example, consider the following recursive rule:

$$\mathsf{T(x,y) :\text{-}\ up(x,z), T(z,w), down(y,\ w).}$$

Suppose that the nested loop join visits the predicates in the same order as they appear in the above rule: up,T,down. In this case, we would build a B-tree on T with search key the first attribute, and a B-tree on down with search key its second attribute. Since there are multiple rules in a Datalog program, it is possible that there is overlap between such indices, for example in the case where the search key of the one B-tree is a prefix of the search key of the other B-tree. Soufflé finds an appropriate subset of indices that "covers" all necessary B-trees to speed up the nested loop joins, and constructs all of them before evaluation starts. Even though index recommendation is generally intractable (Piatetsky-Shapiro, 1983), in this more restrictive setting it can be solved in polynomial time.

We should note here that this design choice by Soufflé means that it can construct multiple indexes for the same relation. Hence, although performance can substantially improve, the memory footprint can become very large, which is prohibitive in a main memory setting.

5

Optimizations

In this section, we present optimization techniques implemented in Datalog systems to further speed up evaluation. We distinguish these optimizations into two categories: *language-level* optimizations that attempt to rewrite the Datalog program into a more efficient program, and *low-level* optimizations that operate on the logical and physical operators of the translated Datalog program.

5.1 Language-level Optimizations

This section discusses optimizations that happen on the language level through different rewriting techniques.

5.1.1 The Magic-Set Transformation

A standard optimization for Datalog programs is the *magic set* transformation (Beeri and Ramakrishnan, 1987). We should remark here that the magic sets rewriting technique has also been applied in the context of non-recursive computation, and in particular to evaluate more efficiently correlated subqueries: this method is known as magic decorrelation (Seshadri *et al.*, 1996).

Some systems (Soufflé, SociaLite) implement this transformation (or a restricted version of it), while other systems do not. From a high level, this technique can be thought of as the equivalent of pushing a selection condition in a query plan, albeit now it happens within a recursive program. For example, consider the following Datalog program:

```
T(x,y) :- Edge(x,y).
T(x,y) :- T(x,z), Edge(z,y).
  A(y) :- T(a,y).
```

This program computes all the nodes in a graph reachable from node a. Note that the program is essentially a recursion followed by a selection operator. Semi-naïve evaluation will compute all tuples in T before applying selection, hence performing redundant computation. The equivalent rewritten program produced by the magic-set transformation is as follows:

```
Tbf(x,y) :- Ti(x), Edge(x,y).
  T(x,y) :- Ti(x), Tbf(x,z), Edge(z,y).
   Ti(x) :- Tbf(x,y).
   Ti(a) :- .
    A(y) :- Tbf(a,y).
```

For the above program, semi-naïve evaluation will only compute the necessary facts because the selection has been pushed inside the recursion through the predicate Ti. Hence, the evaluation will be much more efficient. We will not cover magic sets in more depth here, and we refer the reader to another survey by Green *et al.* (2013) for an in-depth presentation of the technique.

5.1.2 Reducing the Number of Iterations

As we discussed in earlier sections, in parallel systems with synchronous computation, the number of iterations determines the number of synchronization barriers during computation. Hence, reducing the number of iterations could speed up parallel evaluation. For instance, linear transitive closure can be rewritten to the following non-linear Datalog program which is guaranteed to terminate in a logarithmic number of iterations (w.r.t. the input size).

```
T(x,y) :- Edge(x,y).
T(x,y) :- T(x,z), T(z,y).
```

Unfortunately, the above program will reduce the number of rounds at the expense of an increase in the number of facts that will be produced. In other words, synchronization will decrease, but communication will increase. In a recent work, Afrati and Ullman (2012) explore whether one can achieve the best of both worlds. They show that some Datalog programs can be evaluated with a logarithmic number of rounds and no order-of-magnitude increase in the number of facts deduced. Transitive closure is an example of such a program. On the other hand, for other (even linear) Datalog programs, this is not possible. For example, reachability and same-generation do not admit such an effective rewriting. This technique, even though promising, has not been implemented yet as part of a Datalog system.

5.1.3 Reducing the Number of Strata

In parallel asynchronous evaluation, we noted that the presence of (stratified) negation introduces a synchronization barrier that is unavoidable. In particular, the number of strata determines the number of synchronization barriers a system needs to impose. A few techniques have been proposed to rewrite the Datalog program so that it reduces the number of strata while computing the same output (Rudolph and Thomazo, 2016; Ketsman and Koch, 2020). The key idea is to combine an "unrolling" of the recursion in the program with checks on whether any subset of negated atoms can be safely removed without changing the semantics. These techniques also remain theoretical and have not yet been incorporated in a system.

5.1.4 Pushing Aggregation into Recursion

In the case of aggregation, rewritings can also help with accelerating the recursive computation. This idea was first explored in the early 1990s

(Sudarshan and Ramakrishnan, 1991) with a focus on max, min, and k largest values. A more recent line of work (Zaniolo *et al.*, 2016; 2017) has looked into how one can push certain types of aggregation (in particular max and min) inside the recursion, in a similar way to how one can push selection through recursion with the magic-set transformation. To explain this idea, consider the following program, which computes the shortest paths from a starting node a:

```
      Path(x, d) :- Edge('a', x, d).
      Path(x, d) :- Path(y, d'), Edge(y, x, d"), d = d' + d".
SPath(x, min(d)) :- Path(x, d).
```

This program will compute all paths from node a, and then compute the minimum. Instead, one can push the aggregation into the recursion by replacing the head of the first two rules with Path(x, min(d)). This introduces aggregation in the recursion, but improves the performance because at every iteration only the shortest paths are stored (instead of all paths). The authors in Zaniolo *et al.* (2016) specify a syntactic property of the program, which they call the PreM property, under which this transformation can be done correctly. We should remark here that a very recent work by Wang *et al.* (2022) has shown how to generalize the PreM transformation – and other rewritings – through the use of program synthesis techniques.

5.2 Low-level Optimizations

In this section, we describe techniques for optimizing Datalog on the logical/physical level.

5.2.1 Extending the Search Space for Plans

The combination of Relational Algebra and recursion can lead to novel ways to optimize query plans. For instance, Jachiet *et al.* (2020) proposes techniques that allow for new execution strategies for a subset of Datalog that is equivalent to RA plus a fixpoint operator. Using this algebraic formulation, the authors come up with novel rewriting rules that generate new query execution plans. A rewrite rule can push a filter inside a fixpoint computation (similar to a magic set transformation), or combine two fixpoints in one single fixpoint computation. To give a concrete example of these strategies, consider the following Datalog program:

```
T(x,y) :- R(x,y).
T(x,y) :- T(x,z), R(z,y).
U(x,y) :- T(x,z), S(z,y).
U(x,y) :- U(x,z), S(z,y).
```

This program computes all pairs of nodes connected by a path that consists of R-edges followed by S-edges. The standard plan here is to first compute T, and then use T to compute U (since T does not depend on U). A different strategy would be to first compute the join $\pi_{x,y}(R(x,z) \bowtie S(z,y))$ as the base facts for U, and then add new facts by expanding left using R or right using S. This strategy cannot be captured by semi-naïve evaluation. In Jachiet *et al.* (2020), a simple optimizer with cost-based estimation determines which of the generated plans performs the best and uses this to evaluate the program. Although this idea has been verified experimentally, it has not been yet incorporated to any general Datalog system.

5.2.2 Delta Stepping

SociaLite uses the *delta stepping* technique (Meyer and Sanders, 1998) to accelerate computation in the presence of recursive monotone aggregate functions. This technique was originally applied to speed up parallel shortest path computation, and it was generalized to linear Datalog programs. To give an example, suppose the aggregate function is a min. In this case, convergence will be faster if the execution algorithm operates with the smallest values. However, since SociaLite uses asynchronous computation, making this choice will hurt parallelism. Delta stepping overcomes this issue by appropriately choosing multiple smallest values to update at once.

6

Conclusion

In this monograph, we presented the recent advancements of Datalog engines to handle large datasets and complex programs across various domains. We particularly focused on the different language variants of Datalog, on how Datalog evaluation can be parallelized, both in a multicore and shared-nothing parallel setting, and how different design choices are necessary for different domains and input data.

Although there has been a lot of progress and success both in the academic and commercial world, there are several exciting research directions, both from a theoretical and practical viewpoint. We discuss next some of these directions.

Approximate Computation. Most Datalog engines compute the result of the program exactly. However, in some applications it would be interesting to trade off accuracy for performance. For example, a user may not be interested in all the output results but only a subset of them (e.g., the user only wants to obtain some of the pairs of nodes that are connected). In the case of aggregate functions (recursive or not), a user may want to see an approximate value of a count, sum, or average. SociaLite (Seo *et al.*, 2013b) supports some form of approximation (by terminating the program before it reaches the fixpoint). A natural question is whether one can obtain approximate answers fast with formal guarantees of accuracy, and whether one can obtain tight trade-offs between runtime and accuracy.

Incremental Computation. Supporting incremental computation is a desirable feature when the input data changes often. For example, consider the case of maintaining the reachable nodes from some starting node in a dynamic graph. DDLog (Ryzhyk and Budiu, 2019) and LogicBlox are the only engines that support some form of incremental computation. However, in the case of DDLog, there is a huge memory cost on creating the necessary data structures to maintain the output efficiently. Is it possible to design data structures and algorithms with better trade-offs between memory consumption and update time?

Asynchronous Computation with Low Communication. Recent theoretical advancements (Ameloot *et al.*, 2013; 2016) characterize which Datalog programs can be correctly computed using asynchronous parallel computation. However, they tell us nothing on whether we can achieve this computation with small communication cost and small computational redundancy. Moreover, as we discussed in Section 3, for some programs it is not clear that an asynchronous strategy is more efficient than the corresponding synchronous one. It is even possible that a mixed strategy of asynchronous and synchronous computation may provide to be the optimal one. Hence, it is natural to pinpoint the classes of Datalog programs that are amenable to fast asynchronous versus synchronous versus mixed strategies, both from a theoretical and practical point of view.

Improved Optimization Techniques. As we have discussed in this monograph, most Datalog engines opt for little to no optimization to find a good query plan. The reasons are multiple: (*i*) Datalog programs can grow to become very complex with many rules, (*ii*) the data is dynamic (since the IDBs are incremented after every iteration), and (*iii*) the interaction of recursion with relational operators makes rewriting methods complex to handle. Some progress on this front has been made (Jachiet *et al.*, 2020), but it remains an interesting direction to see how much performance can be improved by better Datalog optimizers that make more aggressive decisions regarding which plan to choose. In particular, it is possible that the iterative nature of Datalog can allow for some form of learning of the input data, which can be used to make better optimization decisions.

Datalog Synthesis. Since Datalog is a high-level declarative language, it provides an excellent opportunity to explore the automatic synthesis of a Datalog program from input/output examples. Some recent efforts (Albarghouthi *et al.*, 2017; Si *et al.*, 2018; Raghothaman *et al.*, 2020) have proposed program synthesis techniques to learn a large set of Datalog programs from examples. Datalog is particularly promising for synthesis since it operates under set semantics (in contrast to SQL). An open question in this area is whether the proposed synthesis techniques can scale to larger and more complex Datalog programs, and whether they can be incorporated in a modern Datalog engine to facilitate user interaction.

New Applications. Recent progress in Consistent Query Answering (CQA) has shown that certain answers for join queries over inconsistent data can be computed through linear Datalog programs (Koutris and Wijsen, 2021). An interesting direction is to explore how current Datalog engines can handle these programs and how they can be optimized to run efficiently.

Another application concerns how Datalog can be extended to support tasks related to Machine Learning (ML). Such a formulation could use the techniques used to speed up and parallelize Datalog to also optimize ML pipelines. In particular, the support for recursive aggregation is a natural fit to how many ML algorithms work, for example, see Gu *et al.* (2019). The challenging task in this case is how one can define the semantics of Datalog such that there is a well-defined output and optimization techniques can be applied without losing the correctness of the computation.

References

ns-3. "Network Simulator 3". URL: http://www.nsnam.org/.

Abadi, M. and B. T. Loo. (2007). "Towards a Declarative Language and System for Secure Networking". In: *NetDB*. USENIX Association.

Abiteboul, S., M. Bienvenu, A. Galland, and É. Antoine. (2011). "A rule-based language for web data management". In: *PODS*. ACM. 293–304.

Abiteboul, S., R. Hull, and V. Vianu. (1995). *Foundations of Databases*. Addison-Wesley.

Afrati, F. N., V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. (2011). "Map-reduce extensions and recursive queries". In: *EDBT*. ACM. 1–8.

Afrati, F. N., S. S. Cosmadakis, and M. Yannakakis. (1995). "On Datalog vs. Polynomial Time". *J. Comput. Syst. Sci.* 51(2): 177–196.

Afrati, F. N. and C. H. Papadimitriou. (1987). "The Parallel Complexity of Simple Chain Queries". In: *PODS*. ACM. 210–213.

Afrati, F. N. and J. D. Ullman. (2010). "Optimizing joins in a map-reduce environment". In: *EDBT*. Vol. 426. ACM. 99–110.

Afrati, F. N. and J. D. Ullman. (2012). "Transitive closure and recursive Datalog implemented on clusters". In: *EDBT*. ACM. 132–143.

Agrawal, S., S. Chaudhuri, and V. R. Narasayya. (2000). "Automated Selection of Materialized Views and Indexes in SQL Databases". In: *VLDB*. Morgan Kaufmann. 496–505.

Albarghouthi, A., P. Koutris, M. Naik, and C. Smith. (2017). “Constraint-Based Synthesis of Datalog Programs”. In: *CP*. Vol. 10416. *Lecture Notes in Computer Science*. Springer. 689–706.

Alvaro, P., N. Conway, J. M. Hellerstein, and W. R. Marczak. (2011). “Consistency Analysis in Bloom: a CALM and Collected Approach”. In: *CIDR*. 249–260.

Alvaro, P., W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. (2010). “Dedalus: Datalog in Time and Space”. In: *Datalog*. Vol. 6702. *Lecture Notes in Computer Science*. Springer. 262–281.

Ameloot, T. J., B. Ketsman, F. Neven, and D. Zinn. (2016). “Weaker Forms of Monotonicity for Declarative Networking: A More Fine-Grained Answer to the CALM-Conjecture”. *ACM Trans. Database Syst.* 40(4): 21:1–21:45.

Ameloot, T. J., F. Neven, and J. V. den Bussche. (2013). “Relational transducers for declarative networking”. *J. ACM*. 60(2): 15:1–15:38.

Aref, M., B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. (2015). “Design and Implementation of the LogicBlox System”. In: *SIGMOD Conference*. ACM. 1371–1382.

Bancilhon, F. and R. Ramakrishnan. (1986). “An Amateur’s Introduction to Recursive Query Processing Strategies”. In: *SIGMOD Conference*. ACM Press. 16–52.

Beame, P., P. Koutris, and D. Suciu. (2017). “Communication Steps for Parallel Query Processing”. *J. ACM*. 64(6): 40:1–40:58.

Beeri, C. and R. Ramakrishnan. (1987). “On the Power of Magic”. In: *PODS*. ACM. 269–284.

Bellomarini, L., E. Sallinger, and G. Gottlob. (2018). “The Vadalog System: Datalog-based Reasoning for Knowledge Graphs”. *Proc. VLDB Endow.* 11(9): 975–987.

Bembenek, A., S. Chong, and M. Gaboardi. “AbcDatalog”. URL: http://abcdatalog.seas.harvard.edu/.

Bollig, B. and I. Wegener. (1996). “Improving the Variable Ordering of OBDDs Is NP-Complete”. *IEEE Trans. Computers*. 45(9): 993–1002.

Bravenboer, M. and Y. Smaragdakis. (2009). "Strictly declarative specification of sophisticated points-to analyses". In: *OOPSLA*. ACM. 243–262.

Brin, S. and L. Page. (1998). "The Anatomy of a Large-Scale Hypertextual Web Search Engine". *Comput. Networks*. 30(1-7): 107–117.

Cabibbo, L. (1995). "On the Power of Stratified Logic Programs with Value Invention for Expressing Database Transformations". In: *ICDT*. Vol. 893. *Lecture Notes in Computer Science*. Springer. 208–221.

Calì, A., G. Gottlob, T. Lukasiewicz, B. Marnette, and A. Pieris. (2010). "Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications". In: *LICS*. IEEE Computer Society. 228–242.

Ceri, S., G. Gottlob, and L. Tanca. (1989). "What you Always Wanted to Know About Datalog (And Never Dared to Ask)". *IEEE Trans. Knowl. Data Eng.* 1(1): 146–166.

Chimenti, D., R. Gamboa, R. Krishnamurthy, S. A. Naqvi, S. Tsur, and C. Zaniolo. (1990). "The LDL System Prototype". *IEEE Trans. Knowl. Data Eng.* 2(1): 76–90.

Conway, N., W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. (2012). "Logic and lattices for distributed programming". In: *SoCC*. ACM. 1.

Cosmadakis, S. S. and P. C. Kanellakis. (1986). "Parallel Evaluation of Recursive Rule Queries". In: *PODS*. ACM. 280–293.

Dean, J. and S. Ghemawat. (2004). "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI*. USENIX Association. 137–150.

Derr, M. A., S. Morishita, and G. Phipps. (1994). "The Glue-Nail Deductive Database System: Design, Implementation, and Evaluation". *VLDB J.* 3(2): 123–160.

DeWitt, D. J. and J. Gray. (1992). "Parallel Database Systems: The Future of High Performance Database Systems". *Commun. ACM*. 35(6): 85–98.

Eisner, J. and N. W. Filardo. (2011). “Dyna: Extending Datalog For Modern AI”. In: *Datalog Reloaded*. Ed. by O. de Moor, G. Gottlob, T. Furche, and A. Sellers. Vol. 6702. *Lecture Notes in Computer Science*. Springer. 181–220.

Fagin, R., P. G. Kolaitis, R. J. Miller, and L. Popa. (2003). “Data Exchange: Semantics and Query Answering”. In: *ICDT*. Vol. 2572. *Lecture Notes in Computer Science*. Springer. 207–224.

Fan, Z., J. Zhu, Z. Zhang, A. Albarghouthi, P. Koutris, and J. M. Patel. (2019). “Scaling-Up In-Memory Datalog Processing: Observations and Techniques”. *Proc. VLDB Endow.* 12(6): 695–708.

Ganguly, S., A. Silberschatz, and S. Tsur. (1990). “A Framework for the Parallel Processing of Datalog Queries”. In: *SIGMOD Conference*. ACM Press. 143–152.

Ganguly, S., A. Silberschatz, and S. Tsur. (1992). “Parallel Bottom-Up Processing of Datalog Queries”. *J. Log. Program.* 14(1&2): 101–126.

Graefe, G. (2010). “A survey of B-tree locking techniques”. *ACM Trans. Database Syst.* 35(3): 16:1–16:26.

Greco, S., C. Zaniolo, and S. Ganguly. (1992). “Greedy by Choice”. In: *PODS*. ACM Press. 105–113.

Green, T. J., S. S. Huang, B. T. Loo, and W. Zhou. (2013). “Datalog and Recursive Query Processing”. *Found. Trends Databases.* 5(2): 105–195.

Gu, J., Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo. (2019). “RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark”. In: *SIGMOD Conference*. ACM. 467–484.

Halperin, D., V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. (2014). “Demonstration of the Myria big data management service”. In: *SIGMOD Conference*. ACM. 881–884.

Hellerstein, J. M. (2010). “The declarative imperative: experiences and conjectures in distributed logic”. *SIGMOD Rec.* 39(1): 5–19.

Hoder, K., N. Bjørner, and L. M. de Moura. (2011). “μZ- An Efficient Engine for Fixed Points with Constraints”. In: *CAV*. Vol. 6806. *Lecture Notes in Computer Science*. Springer. 457–462.

Jachiet, L., P. Genevès, N. Gesbert, and N. Layaïda. (2020). "On the Optimization of Recursive Relational Queries: Application to Graph Queries". In: *SIGMOD Conference*. ACM. 681–697.

Jordan, H., P. Subotic, D. Zhao, and B. Scholz. (2019a). "A specialized B-tree for concurrent datalog evaluation". In: *PPoPP*. ACM. 327–339.

Jordan, H., P. Subotic, D. Zhao, and B. Scholz. (2019b). "Brie: A Specialized Trie for Concurrent Datalog". In: *PMAM@PPoPP*. ACM. 31–40.

Kanellakis, P. C. (1986). "Logic Programming and Parallel Complexity". In: *ICDT*. Vol. 243. *Lecture Notes in Computer Science*. Springer. 1–30.

Ketsman, B., A. Albarghouthi, and P. Koutris. (2020). "Distribution Policies for Datalog". *Theory Comput. Syst.* 64(5): 965–998.

Ketsman, B. and C. Koch. (2020). "Datalog with Negation and Monotonicity". In: *ICDT*. Vol. 155. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. 19:1–19:18.

Kifer, M. (1998). "On the Decidability and Axiomatization of Query Finiteness in Deductive Databases". *J. ACM*. 45(4): 588–633.

Kossmann, D. (2000). "The State of the art in distributed query processing". *ACM Comput. Surv.* 32(4): 422–469.

Koutris, P. and J. Wijsen. (2021). "Consistent Query Answering for Primary Keys in Datalog". *Theory Comput. Syst.* 65(1): 122–178.

Lam, M. S., J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. (2005). "Context-sensitive program analysis as database queries". In: *PODS*. ACM. 1–12.

Leone, N., G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. (2006). "The DLV system for knowledge representation and reasoning". *ACM Trans. Comput. Log.* 7(3): 499–562.

Loo, B. T., T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. (2006). "Declarative networking: language, execution and optimization". In: *SIGMOD Conference*. ACM. 97–108.

Loo, B. T., T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. (2009). "Declarative networking". *Commun. ACM*. 52(11): 87–95.

Madsen, M., M. Yee, and O. Lhoták. (2016). "From Datalog to flix: a declarative language for fixed points on lattices". In: *PLDI*. ACM. 194–208.

Malewicz, G., M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. (2010). "Pregel: a system for large-scale graph processing". In: *SIGMOD Conference*. ACM. 135–146.

Marczak, W. R., S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. (2010). "SecureBlox: customizable secure distributed data processing". In: *SIGMOD Conference*. ACM. 723–734.

Martinez-Angeles, C. A., I. de Castro Dutra, V. S. Costa, and J. Buenabad-Chávez. (2013). "A Datalog Engine for GPUs". In: *KDPD*. Vol. 8439. *Lecture Notes in Computer Science*. Springer. 152–168.

Mazuran, M., E. Serra, and C. Zaniolo. (2013). "Extending the power of datalog recursion". *VLDB J.* 22(4): 471–493.

Meyer, U. and P. Sanders. (1998). "Delta-Stepping: A Parallel Single Source Shortest Path Algorithm". In: *ESA*. Vol. 1461. *Lecture Notes in Computer Science*. Springer. 393–404.

Moffitt, V. Z., J. Stoyanovich, S. Abiteboul, and G. Miklau. (2015). "Collaborative Access Control in WebdamLog". In: *SIGMOD Conference*. ACM. 197–211.

Murray, D. G., F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. (2013). "Naiad: a timely dataflow system". In: *SOSP*. ACM. 439–455.

Patel, J. M., H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. (2018). "Quickstep: A Data Platform Based on the Scaling-Up Approach". *Proc. VLDB Endow.* 11(6): 663–676.

Piatetsky-Shapiro, G. (1983). "The Optimal Selection of Secondary Indices is NP-Complete". *SIGMOD Rec.* 13(2): 72–75.

Raghothaman, M., J. Mendelson, D. Zhao, M. Naik, and B. Scholz. (2020). "Provenance-guided synthesis of Datalog programs". *Proc. ACM Program. Lang.* 4(POPL): 62:1–62:27.

Ramakrishnan, R., F. Bancilhon, and A. Silberschatz. (1987). "Safety of Recursive Horn Clauses With Infinite Relations". In: *PODS*. ACM. 328–339.

Ramakrishnan, R., W. G. Roth, P. Seshadri, D. Srivastava, and S. Sudarshan. (1993). "The CORAL Deductive Database System". In: *SIGMOD Conference*. ACM Press. 544–545.

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

RapidNet. "RapidNet Declarative Networking Engine". URL: http://netdb.cis.upenn.edu/rapidnet/.

Reps, T. W. (1993). "Demand Interprocedural Program Analysis Using Logic Databases". In: *Workshop on Programming with Logic Databases (Book), ILPS. The Kluwer International Series in Engineering and Computer Science 296*. Kluwer. 163–196.

Ross, K. A. and Y. Sagiv. (1992). "Monotonic Aggregation in Deductive Databases". In: *PODS*. ACM Press. 114–126.

Rudolph, S. and M. Thomazo. (2016). "Expressivity of Datalog Variants - Completing the Picture". In: *IJCAI*. IJCAI/AAAI Press. 1230–1236.

Ryzhyk, L. and M. Budiu. (2019). "Differential Datalog". In: *Datalog*. Vol. 2368. *CEUR Workshop Proceedings*. CEUR-WS.org. 56–67.

Scholz, B., H. Jordan, P. Subotic, and T. Westmann. (2016). "On fast large-scale program analysis in Datalog". In: *CC*. ACM. 196–206.

Seib, J. and G. Lausen. (1991). "Parallelizing Datalog Programs by Generalized Pivoting". In: *PODS*. ACM Press. 241–251.

Seo, J., S. Guo, and M. S. Lam. (2013a). "SociaLite: Datalog extensions for efficient social network analysis". In: *ICDE*. IEEE Computer Society. 278–289.

Seo, J., S. Guo, and M. S. Lam. (2015). "SociaLite: An Efficient Graph Query Language Based on Datalog". *IEEE Trans. Knowl. Data Eng.* 27(7): 1824–1837.

Seo, J., J. Park, J. Shin, and M. S. Lam. (2013b). "Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis". *Proc. VLDB Endow.* 6(14): 1906–1917.

Seshadri, P., H. Pirahesh, and T. Y. C. Leung. (1996). "Complex Query Decorrelation". In: *ICDE*. IEEE Computer Society. 450–458.

Sewall, J., J. Chhugani, C. Kim, N. Satish, and P. Dubey. (2011). "PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors". *Proc. VLDB Endow.* 4(11): 795–806.

Shaw, M., P. Koutris, B. Howe, and D. Suciu. (2012). "Optimizing Large-Scale Semi-Naive Datalog Evaluation in Hadoop". In: *Datalog.* Vol. 7494. *Lecture Notes in Computer Science.* Springer. 165–176.

Shkapsky, A., M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. (2016). "Big Data Analytics with Datalog Queries on Spark". In: *SIGMOD Conference.* ACM. 1135–1149.

Shkapsky, A., M. Yang, and C. Zaniolo. (2015). "Optimizing recursive queries with monotonic aggregates in DeALS". In: *ICDE.* IEEE Computer Society. 867–878.

Si, X., W. Lee, R. Zhang, A. Albarghouthi, P. Koutris, and M. Naik. (2018). "Syntax-guided synthesis of Datalog programs". In: *ESEC/SIGSOFT FSE.* ACM. 515–527.

Smaragdakis, Y. and G. Balatsouras. (2015). "Pointer Analysis". *Found. Trends Program. Lang.* 2(1): 1–69.

Stonebraker, M. (1985). "The Case for Shared Nothing". In: *HPTS.*

Subotić, P., H. Jordan, L. Chang, A. Fekete, and B. Scholz. (2018). "Automatic Index Selection for Large-Scale Datalog Computation". *Proc. VLDB Endow.* 12(2): 141–153.

Sudarshan, S. and R. Ramakrishnan. (1991). "Aggregation and Relevance in Deductive Databases". In: *VLDB.* Morgan Kaufmann. 501–511.

Ullman, J. D. (1989). *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press.

Ullman, J. D. and A. V. Gelder. (1988). "Parallel Complexity of Logical Query Programs". *Algorithmica.* 3: 5–42.

Veldhuizen, T. L. (2014). "Triejoin: A Simple, Worst-Case Optimal Join Algorithm". In: *ICDT.* OpenProceedings.org. 96–106.

Wang, J., M. Balazinska, and D. Halperin. (2015). "Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines". *Proc. VLDB Endow.* 8(12): 1542–1553.

Wang, K., A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani. (2017). "Graspan: A Single-Machine Disk-Based Graph System for Interprocedural Static Analyses of Large-Scale Systems Code". *SIGARCH Comput. Archit. News.* 45(1): 389–404.

Wang, Y. R., M. A. Khamis, H. Q. Ngo, R. Pichler, and D. Suciu. (2022). "Optimizing Recursive Queries with Program Synthesis". *CoRR.* abs/2202.10390.

Whaley, J. and M. S. Lam. (2004). "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams". In: *PLDI.* ACM. 131–144.

Wolfson, O. (1989). "Sharing the Load of Logic-Program Evaluation". *IEEE Data Eng. Bull.* 12(1): 58–64.

Wolfson, O. and A. Ozeri. (1990). "A New Paradigm for Parallel and Distributed Rule-Processing". In: *SIGMOD Conference.* ACM Press. 133–142.

Wolfson, O. and A. Ozeri. (1993). "Parallel and Distributed Processing of Rules by Data Reduction". *IEEE Trans. Knowl. Data Eng.* 5(3): 523–530.

Wolfson, O. and A. Silberschatz. (1988). "Distributed Processing of Logic Programs". In: *SIGMOD Conference.* ACM Press. 329–336.

Yang, M., A. Shkapsky, and C. Zaniolo. (2017). "Scaling up the performance of more powerful Datalog systems on multicore machines". *VLDB J.* 26(2): 229–248.

Zaharia, M., R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. (2016). "Apache Spark: a unified engine for big data processing". *Commun. ACM.* 59(11): 56–65.

Zaniolo, C., M. Yang, A. Das, and M. Interlandi. (2016). "The Magic of Pushing Extrema into Recursion: Simple, Powerful Datalog Programs". In: *AMW.* Vol. 1644. *CEUR Workshop Proceedings.* CEUR-WS.org.

Zaniolo, C., M. Yang, A. Das, A. Shkapsky, T. Condie, and M. Interlandi. (2017). "Fixpoint semantics and optimization of recursive Datalog programs with aggregates". *Theory Pract. Log. Program.* 17(5-6): 1048–1065.

Zhang, W., K. Wang, and S. Chau. (1995). "Data Partition and Parallel Evaluation of Datalog Programs". *IEEE Trans. Knowl. Data Eng.* 7(1): 163–176.