Logica-TGD: Transforming Graph Databases Logically

Evgeny Skvortsov
Google LLC, Kirkland, WA, USA

Yilin Xia
School of Information Sciences, University of Illinois Urbana-Champaign, IL, USA

Bertram Ludäscher
School of Information Sciences, University of Illinois Urbana-Champaign, IL, USA

Shawn Bowers
Department of Computer Science, Gonzaga University, Spokane, WA, USA

Abstract

Graph transformations are a powerful computational model for manipulating complex networks, but handling temporal aspects and scalability remain significant challenges. We present a novel approach to implementing graph transformations using Logica, an open-source logic programming language and system that operates over standard database systems including PostgreSQL, DuckDB, and BigQuery. Logica leverages the inherent parallelism of these engines and offers a practical and scalable way to carry out a variety of graph transformations, including complex scenarios such as time-varying graphs. We illustrate Logica's graph querying and transformation capabilities with several examples, including a declarative program for pathfinding in a dynamic graph and a taxonomic analysis over a full current Wikidata dump. Experimental results compare Logica running on DuckDB engine to the Datalog engines Soufflé and Nemo, and to DuckPGQ, an implementation of SQL/PGQ. We argue that a logic-based declarative syntax, a built-in visualization library, and (plug-and-play) support for large-scale database engines make Logica a convenient and practical tool for a wide range of graph transformation tasks.

2012 ACM Subject Classification Information systems → Query languages for non-relational engines

Keywords and phrases Logic rules, Graph queries, Graph transformations

Digital Object Identifier 10.4230/TGDK.4.2.7

Category Resource

Related Version *Previous Version*: https://arxiv.org/abs/2503.00568

Supplementary Material *Software (Source Code)*: https://github.com/EvgSkv/logica

Received 2025-07-19 **Accepted** 2026-06-22 **Published** 2026-09-03

Editors Stefania Dumbrava, George Fletcher, Matteo Lissandrini, and Riccardo Tommasini

Special Issue Data Management for (Knowledge) Graphs

Generative AI Declaration Generative AI was used in a limited, assistive capacity in preparing this article, and all output was reviewed and verified by the authors. Generative AI was not used to generate references or citations.

1 Introduction

Graph transformations are a powerful and versatile method for modeling and manipulating complex systems across diverse fields, ranging from software engineering [9, 27] and social network analysis [14] to biology and chemistry [35, 31, 36]. These transformations typically operate by applying rewrite rules to a graph, altering its structure and properties. While such transformations are known for their expressiveness and flexibility, their implementation can often be complex, especially when dealing with time-varying (i.e., evolving) graphs and when scalable solutions are needed. Additionally, existing graph database systems often provide limited support for such evolution mechanisms, creating a gap between theoretical models and practical implementations.

Logic programming, on the other hand, provides a declarative approach to problem solving by expressing relationships using rules rather than explicitly stating control-flow. Logic-based systems (e.g., Prolog and Answer Set Programming [17]) are well-established in various areas, including (symbolic) AI and knowledge representation and reasoning.

More recently, Logica (*Logic* + *A*ggregation) [41, 40], an open-source logic programming language and system, has emerged, which employs performant, parallel data processing environments such as DuckDB and BigQuery. Logica combines the declarative power of logic programming with the scalability and efficiency of modern databases, offering a promising new path for tackling graph transformation challenges.

This paper explores the application of Logica to the domain of graph transformations, bridging the gap between these two paradigms. Our approach leverages Logica's ability to process large datasets in a parallel and efficient manner, providing a novel means of implementing and executing graph transformations at scale using logical rules. We argue that with a graph transformation mindset, logic programming can provide a natural, powerful, and intuitive approach for defining transformations, opening new opportunities within both communities. Crucially, Logica also enables a practical approach to addressing issues that are difficult in classical graph transformation approaches, such as temporal transformations and seamless scalability.

To illustrate the practical nature of our approach, we introduce several example Logica programs implementing different types of graph transformations. Specifically, we show how Logica can be used to define and perform basic graph transformations including message passing, removing transitively implied graph edges, graph extraction, and computing complex annotation properties (by finding solution labels for two-player combinatorial game graphs). We also introduce a novel approach to time-aware pathfinding (e.g., [25, 8]), directly addressing the need for principled and tractable mechanisms to handle evolving graph data, a notoriously difficult problem in classical graph transformation approaches. These examples serve both to show the expressiveness of our approach, and to illustrate its potential benefits.

In the experimental evaluation (Section 4), we include a same-generation benchmark where aggregation within recursion enables a purely declarative solution: the generation of a child is defined as the next generation after its parent's, and same-generation pairs follow by equality. This illustrates how Logica's support for aggregation in recursive rules extends the class of problems for which a declarative definition also serves as an efficient algorithm.

Examples demonstrated in this paper are available online [39], as part of the Logica repository [38]. Examples are represented in online Google Jupyter Notebooks (via Colab), where they can be executed and modified.

The rest of this paper is organized as follows. Section 2 provides an overview of Logica including its overall architecture, syntax and extensions (compared to traditional Datalog), and its program evaluation approach. Section 3 demonstrates how to express a variety of graph transformations in Logica. In Section 4 we report results of comparison of Logica execution speed with Soufflé, Nemo and DuckPGQ engines. And in Section 5 we describe location of Logica in the body of related work. Section 6 concludes with a discussion of limitations and potential avenues for future work.

2 Logica Overview

Logica is a freely available, open-source logic programming language that bridges declarative rule specification with the computational power of modern parallel databases. Rather than implementing a specialized engine, Logica employs a compilation-based architecture that translates logical programs into optimized SQL, enabling execution on industrial database systems DuckDB, SQLite, PostgreSQL, and BigQuery.

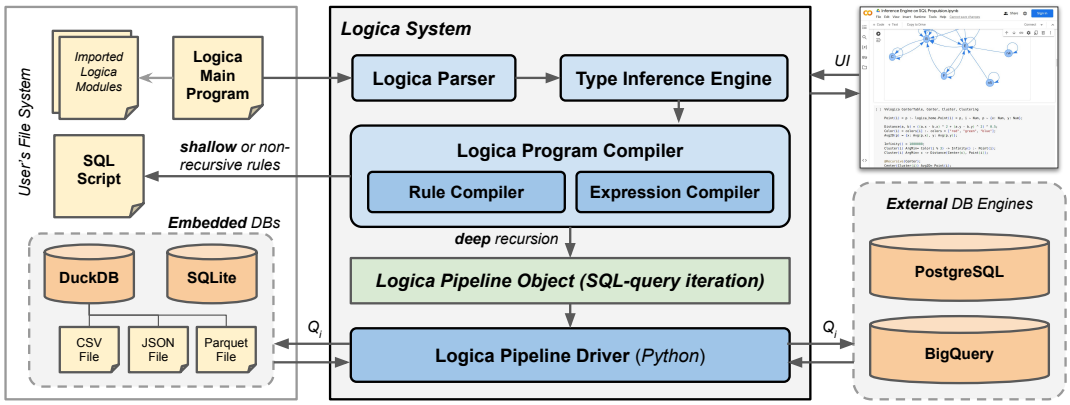

Figure 1 System Architecture: Logica supports a module system for importing libraries (top left); users write programs either locally (top left), e.g., as part of a larger Python script, or from within Jupyter notebooks (top right); programs are parsed, type checked (top middle), and, if well-formed, compiled to a self-contained SQL script (no/limited recursion) or to an iterative plan of pipelined SQL-queries that is passed to a pipeline driver (bottom middle). Logica then executes and monitors queries via underlying SQL engines, either locally (bottom left) or externally (bottom right).

2.1 Logica System Architecture

The Logica compilation process, illustrated in Figure 1, consists of three main stages. The *parser* reads the source program and build an Abstract Syntax Tree (AST). The *type inference engine* analyzes this tree, assigning types for error detection and dialect-specific SQL generation. Finally, the *compiler* translates the type-annotated program into executable SQL.

For non-recursive or shallow-recursive programs, the compiler produces a single, self-contained SQL script. For complex recursive computations, which are common in graph transformation, Logica generates SQL that unfolds recursion into a fixed number of iterations using intermediate tables or nested queries. Each iteration applies the recursive rule to results from the previous iteration, with results combined via UNION ALL. This iterative unfolding mechanism is fundamental to Logica's ability to handle the complex transformations central to graph processing. For programs requiring dynamic fixpoint computation, Logica can generate a Python-based pipeline that iteratively executes these SQL queries until convergence or a termination condition is met.

Logica is capable of using several SQL engines, running its pipeline on SQLite, DuckDB, PostgreSQL, or BigQuery. SQLite is a light yet efficient engine that comes along with Python; with it you can run Logica without installing anything else, however you need to enable it explicitly in your program with an `@Engine("sqlite")` directive. The default engine for Logica is DuckDB, a highly efficient engine built specifically for massive data analysis. DuckDB achieves such efficiency because massive data processing and analysis is its primary purpose, while SQLite and PostgreSQL have much of their focus on efficiently supporting individual-level transactions; thus DuckDB is more aligned with Logica's purpose. Finally, when data that needs processing is of terabytes in volume, you can process it with Logica using BigQuery, a massively parallel data warehouse.

Logica is distributed via PyPI and can be installed with `pip install logica`. It can be used from the command line or from Jupyter notebooks. Documentation and source code are available at logica.dev and github.com/EvgSkv/logica. The compiler architecture is modular with respect to the SQL backend; for instance, experimental support for ClickHouse and Trino has been explored. Supporting a new engine requires accommodating its SQL dialect and available features.

2.2 Logica Syntax Extensions

Logica extends traditional Datalog with several syntactic features designed to help with readability and robustness, particularly for applications involving structured relations and graph transformations.

Named and Positional Arguments. Logica supports both traditional positional arguments and named arguments. Positional arguments define predicates through variable order, suitable for simple relationships. The following example defines two facts and a single rule in Logica using positional arguments. In Logica, predicates begin with uppercase characters, variables begin with lowercase characters, and statements (facts and rules) are terminated by semicolons.

```
Parent("Adam", "Cain");
Parent("Cain", "Enoch");
Grandparent(x, z) :- Parent(x, y), Parent(y, z);
```

Disjunction in Logica, like in Prolog, can be expressed with multiple rules for one predicate, or with the binary operator `|`. Conjunction is of higher priority than disjunction, so parentheses are not required around individual disjuncts. For example, the two facts above can equivalently be written as a single rule:

```
Parent(x, y) :- x == "Adam", y == "Cain" |
                x == "Cain", y == "Enoch";
```

Named arguments, as in SQL, can help with readability, especially for relations with many attributes. The following example shows an Employee relation with four named attributes representing the employee's id, name, department, and salary.

```
Employee(employee_id: 101, name: "Alice",
         department: "Sales", salary: 80000);
Employee(employee_id: 102, name: "Bob",
         department: "Engineering", salary: 95000);
```

Named arguments provide position independence and robustness to schema evolution: queries remain valid when attributes are added, removed, or reordered, as they only reference the attributes they need. The following example queries for names of employees in the engineering department, referencing only the name and department attributes.

```
Engineer(name:) :-
  Employee(name:, department: "Engineering");
```

As shown, in Logica the named attribute can be used in place of the corresponding variable (where the variable name is assumed to be the same as the named attribute).

Logica allows using named attributes directly as variables when the variable name matches the attribute name, as shown with (employee) "`name:`" above.

Positional arguments are internally treated as named arguments `col0`, `col1`, etc. A predicate always corresponds to a single table regardless of how many arguments are mentioned in a particular rule. When arguments are omitted, Logica does not constrain or project on those columns. For example, if we have `Parent(x, y)` and `Grandparent(x, y)` predicates where `x` is a parent (grandparent) of `y`, and we want to find people who are parents or grandparents, we can write:

```
Invite(x) :- Parent(x) | Grandparent(x);
```

Here `Parent(x)` refers to the same table as `Parent(x, y)`, with the second argument simply left unconstrained.

Composite Data Types. Logica supports structured data through lists and records, enabling natural representation of complex relationships common in graph data. Lists in Logica are ordered collections and can contain any type of element. Lists are defined using square bracket notation and Logica supports list iteration and membership testing. The following gives a simple example demonstrating lists and membership checks for an input predicate `Country` containing a list of cities for each country.

```
# Membership test using a list literal
European(city) :- city in ["Paris", "London", "Berlin"];

# Iterate over list elements to create a new relationship
CountryCity(country:, city:) :-
  Country(name: country, cities:), city in cities;
```

Records provide structured access to related data through named fields, similar to objects or structs in other languages. The following example defines two books each containing a nested record and a corresponding query using path expressions to obtain the nested record information.

```
# Define two tuples containing nested records
Book(title: "To Kill a Mockingbird",
     info: {author: "Harper Lee", publication_year: 1960});
Book(title: "1984",
     info: {author: "George Orwell", publication_year: 1949});

# Access record fields using dot notation
RecentBook(title:) :-
  Book(title:, info:),
  info.publication_year > 1950;
```

Records and lists can be arbitrarily nested in Logica, which provides greater flexibility in modeling data as well as for interoperating with external systems and data sources (e.g., that leverage JSON encodings). Lists and records are also useful for representing property graphs in which nodes and edges can be annotated with complex attribute values.

Aggregation. Like SQL, Logica supports aggregation operations. Logica provides two aggregation modes: *predicate-level* and *inline*. The following is an example of predicate-level aggregation. For each author that has written a book, the query computes the number of books written as well as the most recent year the author published a book.

```
# Number of books and most recent publication year by author
AuthorStats(author:,
            book_count? += 1,
            latest_year? Max= publication_year) distinct :-
  Books(author:, publication_year:);
```

The `distinct` keyword is used to trigger aggregation in Logica, indicating the desire to create a set of objects. The columns to be aggregated are denoted with `?` after which an aggregation operator is given followed by `=`; other columns are not aggregated. The set is created for keys

composed of values of non-aggregated columns. Values of aggregated columns are grouped for each key and the aggregation operator is applied to the group. In the example above, the result is one unique row per author. Since multiple books may have the same author, Logica naturally groups these together and applies the aggregation functions (`+=` for counting, `Max=` for latest year) within each group.

Note that whenever a SQL statement has a `GROUP BY` clause, the resulting table has no duplicates, i.e., all rows in the table are *distinct*. In Logica, the `distinct` keyword in the head of the rule serves this same purpose, unifying the concept of sets with aggregation.

Thus while Logica inherently uses multi-set semantics, set semantics organically emerges from aggregation. Denoting a predicate that has no columns aggregated, `distinct` makes it a set, in accordance with aggregation as defined above. Note that similarly in SQL, while there is a `DISTINCT` keyword, the same effect can be achieved via grouping by all columns.

When multiple rules define the same predicate, their contributions are combined via `UNION ALL` and the aggregation operator is then applied over the combined result, which is expressed in SQL as `UNION ALL` wrapped in `GROUP BY`. Logica requires all rules for a predicate to use a consistent aggregation signature, and inconsistent signatures result in a compilation error.

In-line aggregation allows aggregation in the body of the predicate. It uses syntax similar to set-builder notation. The following example uses inline aggregation to compute the total number of special-offer points awarded for each shopping cart where a shopping cart consists of a list of item names and each special offer relates an item name to a record containing its corresponding number of bonus points.

```
CartBonusPoints(cart_id:, total_points:) :-
  ShoppingCart(cart_id:, items:),
  total_points = Sum{
    bonus_points :-
      item_name in items,
      SpecialOffers(item_name:, bonus_points:)
  };
```

In-line aggregation can also construct lists in the body of rules (a form of list comprehension) using the `List` aggregating operator. In the following example, `prices_of_premium_items` is computed assuming that each `item` is a record that has fields `is_premium` and `price`.

```
PremiumCarts(cart_id:, prices_of_premium_items:) :-
  ShoppingCart(cart_id:, items:),
  prices_of_premium_items = List{
    item.price :-
      item in items,
      item.is_premium == true
  };
```

Note that `List` aggregation is non-deterministic: the order of elements in the resulting list depends on race conditions of parallel data processing, just like `array_agg` in PostgreSQL and `list` in DuckDB.

In-line aggregation expressions distinguish two kinds of variables by their scope. Variables that appear both inside and outside the aggregation are *correlated*: their values are fixed by the outer context, just as in SQL correlated subqueries. Variables that appear only inside the aggregation are *existential*: they range over all matching values and are aggregated over. The same scoping rule applies to negation: in `~P(x, y)`, a variable bound in the outer rule body is correlated, while a variable appearing only inside the negation is existentially quantified. Aggregation expressions can be nested. In this case, a variable mentioned in an enclosing aggregation scope is correlated in the inner one.

Functional Notation. Logica bridges declarative logic and functional programming by allowing predicates to be defined and used as functions. Internally, functional values are stored in a special attribute `logica_value`. The syntax `F(x) = y` provides a syntactic shorthand for predicate definitions. The following provides an example of the (implicit) functional style provided in Logica and the corresponding (explicit) definition using the `logica_value` attribute.

```
# Defining a square operation in Functional style:
Square(x) = x * x;

# Defining square using an explicit relational definition:
Square(x, logica_value: x * x);
```

When predicates are used functionally, Logica performs semantic "desugaring." For example, the following example uses the square operation in its functional form followed by the corresponding "desugared" syntax.

```
# Using the square operation (implicitly)
Q(x, Square(x)) :- T(x);

# An equivalent definition with square used in the body of the rule:
Q(x, y) :- T(x), Square(x, logica_value: y);
```

One use of functional predicates is for lookup tables, as shown in the following example.

```
# Set the captial of France and Japan
CountryCapital("France") = "Paris";
CountryCapital("Japan") = "Tokyo";

# Obtain the capital of France
FrenchCapital(CountryCapital("France"));
```

Functional predicates can also be defined using aggregation operators, where the functional value itself is computed by aggregating over related data. The following example computes the total population of each country using the functional value of `CityPopulation` (which gives the population value of a given city) where the `Country` relation consists of countries and their corresponding list of cities.

```
# For each country, compute the total population of its cities:
Population(country) += CityPopulation(city) :-
  Country(name: country, cities:), city in cities;
```

The above example defines `Population` as a functional predicate that returns the total population for any given country by summing the populations of its cities. Note that the `distinct` keyword is omitted here: when the aggregated column is the return value of a functional predicate, the aggregation is clearly visible at the definition site, making `distinct` unnecessary. For predicate-level aggregation with `?`, the aggregated columns may be harder to spot among many columns, so `distinct` serves as an explicit marker that set semantics is intended.

Negation as Aggregation. Logica treats logical negation (denoted as `~`) not as a primitive operation but as emerging naturally from aggregation. As a simple example, the following rule finds birds that can sing but cannot fly.

```
SingingNonFlyer(name:) :-
  Bird(name:), CanSing(name:), ~CanFly(name:);
```

The negation operator is syntactic sugar for aggregation-based non-existence checking. Thus, the above rule is equivalent to the following aggregate query in Logica.

```
SingingNonFlyer(name:) :-
  Bird(name:), CanSing(name:), Sum{1 :- CanFly(name:)} is null;
```

The expression `Sum{1 :- CanFly(name:)}` counts the number of tuples where the bird can fly. Empty sets yield `null` aggregation results, checked with standard `is null` operators. This approach unifies negation with aggregation principles, simplifying the language's conceptual foundation.

2.3 Value Creation

Logica supports string and number manipulation, including arithmetic operations and string concatenation. These operations can be used to create new values, playing the same role as Skolemization. For instance, the following is an example of basic graph reification, where edges become nodes in the given graph.

```
EdgeNode(x, y) = "Edge:" ++ ToString(x) ++ "-to-" ++ ToString(y);
Edge(x, EdgeNode(x, y)) :- DirectEdge(x, y);
Edge(EdgeNode(x, y), y) :- DirectEdge(x, y);
```

Value creation [2] can easily lead to programs without a finite fixpoint. Defining semantics for such programs is more challenging than for finite-domain Datalog, where well-founded semantics offers an iterative solution and stable model semantics reduces to search within a finite space. With value creation, the search space itself may be infinite, and the question of whether evaluation terminates can be undecidable [19]. In practice, Logica uses a simple iterative evaluation strategy, repeating evaluations up to a fixed threshold or until a stopping condition is met; see Section 2.4 for further details. As one example, the simple reification program above does not use recursion, and thus terminates after a single rule application. The following Logica program, however, does not have a finite fixpoint, and its naive bottom-up evaluation will not terminate. By default, Logica executes programs under a limit of 32 iterations, which can be modified by the user via a language directive.

```
NextHop(x) = "next-" ++ ToString(x);
Path(x, NextHop(x)) :- Path(x, something);
```

2.4 Logica Evaluation Strategy

Understanding how Logica executes graph transformation programs provides essential insight for writing effective code. This section describes Logica's systematic approach to evaluating programs with complex dependencies and recursion.

Dependency Analysis. Program evaluation in Logica begins by analyzing predicate dependencies to determine the order of rule evaluation.

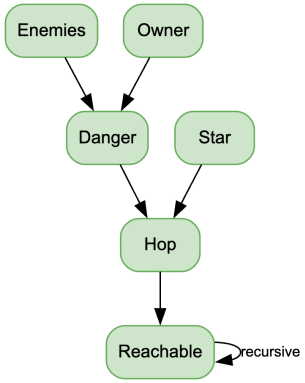

Figure 2 Dependencies $P \to Q$ between predicates P and Q, stating that P is used in the body of a rule to define Q, for the example program of Section 2.4. The predicate Reachable is recursive and iterated either a fixed number of times or until a user-specified stopping condition is met (where the two different approaches are specified by the user as part of the program configuration).

Consider the following example. The predicate `Reachable` is assumed to hold pairs of `Star`s such that a spaceship can `Hop` from star to star if there does not exist a `Danger` in performing the hop. A `Danger` is present if the `Owners` of the stars are `Enemies` of each other. In this example, `Star`, `Owner`, and `Enemies` are given as inputs to the program (i.e., form the EDB).

```
Danger(a, b) :- Enemies(Owner(a), Owner(b));

Hop(a, b) :- Star(a), Star(b), ~Danger(a, b);

Reachable(a, b) distinct :- Hop(a, b);
Reachable(a, b) distinct :- Reachable(a, x), Reachable(x, b);
```

The `distinct` keyword in the `Reachable` rules ensures set semantics: without it, the multiplicity of each pair would equal the number of paths connecting them, which is infinite in the presence of cycles. The corresponding predicate dependencies of the program are shown on Figure 2

Strongly Connected Components. Prior to program execution, Logica uses predicate dependencies to construct a dependency graph from which strongly connected components (SCCs) are identified. Each SCC represents a group of predicates that have dependencies on each other, either directly or indirectly through a graph cycle. If a predicate is not involved in recursion then it forms its own connected component. In the above example, each predicate is a separate connected component and the predicate `Reachable` is the only recursive predicate.

Components in Logica are evaluated in dependency order: a component cannot be processed until all of the components it depends on have been evaluated (following a typical "bottom-up" evaluation strategy). The dependency graph tracks both positive and negative dependencies between predicates. When negation is applied only to predicates from earlier components, the program is stratified and the component-wise evaluation naturally yields the standard stratified semantics (see Section 2.5.2). In the example, the derived predicates `Danger`, `Hop`, and `Reachable` (i.e., forming the IDB) start as empty relations. After the first iteration, `Hop` holds all pair of

(non-`Danger`) stars. The second iteration results in `Reachable` holding for all such pairs of stars. In subsequent iterations, `Reachable` becomes "self-sustaining". We note that components without recursion are evaluated once, as everything that they depend on is already computed.

Handling Recursion. When evaluating recursive components, Logica uses an iterative approach:

- **Start** with base cases established by non-recursive rules.
- **Iterate**:
  - Apply all rules – both base and recursive – using the previous iteration's state computing a new set of facts that hold.
  - Replace the previous state of each relation entirely with those new facts.
  - **Terminate** after a fixed number of iterations (32 by default on DuckDB) or when an explicit stopping condition is met.

This process builds up complete solutions step by step. For reachability, it discovers connections of length 1, then length 2, then length 3, and so on, until all possible connections are found. Also note that fact remains in the state on the next step only if it was re-derived with the facts of the state. Thus if a fact is no longer supported by the state it disappears and thus non-monotonicity is achieved. Hence when we compute distances a derived fact about a distance between two nodes disappears if a shorter distance is found.

Termination and Control. Logica continues evaluating predicates until a stopping condition is met. Logica provides two ways to control iteration via arguments of `@Recursive` annotation of the corresponding predicate.

Fixed number of iterations can be specified if you know from the size of the graph the number of iterations required. Generally we expect algorithm to converge to a fixed point and thus an upper bound of possibly needed number of iterations can be used.

```
@Recursive(Reachable, 20);
Reachable(a, b) distinct :- Hop(a, b);
Reachable(a, b) distinct :- Reachable(a, x), Reachable(x, b);
```

Default behavior. By default when neither the number of iterations nor explicit stopping conditions are specified the iteration proceeds for 32 steps.

Although a small number of iterations, the default has proven useful as an initial starting point for iteration. For example, for many smaller-sized graphs, 32 steps is often sufficient to converge to a fixpoint. Additionally, double-recursion algorithms usually converge with exponential speed [44]. The implementation of reachability with a linear recursion in 32 steps would finds paths of length up to 32. However, as shown in the above example, where reachability is implemented through double recursion, the length of the explored path grows exponentially with the number of iterations, and thus, 32 steps would suffice even for very large graphs.

Explicit stopping conditions can be specified in Logica. In this case, iteration will continue until the stopping condition is met.

```
@Recursive(Reachable, -1, stop: ReachabilityComplete);
Reachable(a, b) distinct :- Hop(a, b);
Reachable(a, b) distinct :- Reachable(a, x), Reachable(x, b);
ReachabilityComplete() distinct :-
  (Reachable(a, x), Hop(x, b)) => Reachable(a, b);
```

The implication `P(x, y) => R(x, z)` is shorthand for `~(P(x, y), ~R(x, z))`, meaning "there's no case where `P` holds but `R` does not." Thus `ReachabilityComplete()` holds when every extendable path has already been extended. The explicit stopping condition can be combined with an upper bound on the number of iterations, or the number of iterations can be given as -1, meaning iteration proceeds indefinitely until the stopping condition is met.

A natural question is whether Logica could provide automatic fixpoint detection, iterating until the state no longer changes without requiring an explicit stopping predicate. While this would sometimes be convenient, in many cases a user-specified stopping condition remains preferable for efficiency, for instance, comparing fact counts is cheaper than comparing entire relation states. Moreover, automatic fixpoint detection would require comparing relation states between iterations, a substantial addition to the current execution model where everything that is computed corresponds directly to a user-defined predicate. Automatic fixpoint detection is a subject of future work.

Power and Limitations. Logica's expressive power and limitations both stem from its execution strategy. Each iteration fully leverages the SQL ecosystem, e.g., if the backend is a distributed engine like BigQuery, a single iteration executes in parallel across the cluster. As SQL engines continue to improve, Logica benefits automatically. However, Logica does not optimize the flow between iterations; it does not implement techniques such as semi-naive evaluation that avoid redundant recomputation across iterations. Consequently, Logica performs well when problems can be solved in a small number of iterations. Fortunately, many practical graph problems fall into this category: transitive closure can be computed in logarithmic iterations using squared recursion, and single-source shortest paths on social networks typically converge within six iterations due to the small-world property. Problems requiring tens of thousands of iterations, however, may not be well-suited for Logica. Additionally, Logica does not implement combinatorial search strategies like those found in Answer Set Programming systems such as Clingo, making it a less suitable syntax for solving NP-complete problems that require exploring exponentially many candidate solutions.

2.5 Logica Semantics

2.5.1 Logica and *While* Semantics

In the previous section, we described Logica's evaluation strategy. From a theoretical perspective, this strategy aligns with the **while** semantic framework introduced by Abiteboul, Hull, and Vianu [1] for database query languages, including **while**(φ), where iteration continues as long as a logical condition φ is satisfied.

The Logica evaluation loop works similarly but uses an **until** loop, i.e., using a condition of termination rather than a condition of continuation. To execute a **while**(φ) evaluation of a recursive predicate P in Logica, a `@Recursive(`P`, -1, stop:` S`)` directive is given as a program annotation, where a predicate S defines the user-specified termination condition $\neg\varphi$. An example using this annotation (with `ReachabilityComplete` as the stopping predicate) is given in Section 2.4.

While Logica execution follows **while** semantics, data types of Logica are richer than classical Datalog. Notably, Logica supports composite values of unbounded size, including lists and records with arbitrarily nested structures, combined with arbitrary arithmetic operations and comparison operators. These additional features enable Turing-complete computation, which establishes computational equivalence with $\mathbf{while}_{\mathbb{N}}$ [1].

2.5.2 Special Cases with Simpler Semantics

Within the while semantics framework, different classes of Logica programs admit less expressive semantic characterizations.

Classical Datalog Semantics. For programs without aggregation or negation, Logica's evaluation reduces to standard least-fixpoint semantics. The iterative process within each strongly connected component corresponds exactly to the well-known immediate consequence operator, ensuring compatibility with traditional Datalog results.

Stratified Datalog Semantics. A Datalog program is stratified if negation in the rules defining a predicate is applied only to predicates that are outside the same strongly connected component [47]. The component-wise evaluation naturally respects this stratification: each SCC is computed to completion using only positive recursion before proceeding to dependent SCCs that may use negation on the completed results.

Monotonic programs refine predicate values in a consistent direction as they iterate. These programs can be characterized by the existence of a partial order on the predicate values (i.e., on the relations themselves) such that each rule application moves values monotonically along this order [26]. In classic Datalog without aggregation, this order is simply set inclusion – rules can only add facts, never remove them, so each iteration produces a relation that contains all previous facts plus potentially new ones. With monotonic aggregation operators (such as `Min=`, `Max=`, or `+=` with non-negative values), the order is induced by the properties of the specific operator being used. Examples include transitive closure computations and shortest path algorithms.

Non-monotonic programs cannot be characterized by such a consistent directional refinement. Values may fluctuate during iteration without a clear monotonic progression visible from the program structure. A classic example is the Win-Move game solver described in Section 3.3, where move evaluations can change as the algorithm refines its analysis of the game tree, and intermediate values may oscillate before converging to the final solution.

Logica is designed to be a practical programming language consistent with the `while` semantics framework. This foundation provides both the expressiveness needed for complex iterative algorithms and the semantic clarity of well-studied formal models.

This generality requires that the programmer reason about execution and ensure that iteration converges in a reasonable number of steps. The programmer must specify either a fixed iteration count or a stopping condition for termination. In practice, a fixed iteration bound often suffices, as the number of steps needed is typically clear from the problem structure.

3 Transforming Graphs with Logica

In Logica, as in Prolog and Answer-Set Programming, graphs can be naturally represented: Nodes are denoted using unique identifiers and edges are represented as facts about the nodes. Graphs can be represented as predicates with two or more *positional* arguments. For example, a simple directed graph can be represented as a binary relation `E(`*source*`,`*target*`)`, where *source* and *target* are nodes connected by a directed edge. Additional graph data, e.g., node or edge attributes, can be expressed using separate predicates or extra positional (in Logica also *named*) arguments. For instance, a graph whose edges are assigned colors can be represented using a ternary predicate and facts of the form `E(`*source*`,`*target*`,`*color*`)`.

Graph transformations can be implemented by defining new predicates that depend on the original graph: e.g., given a graph with edges `E`, a new graph `E2` extending `E` by adding new edges between nodes that are separated by two "hops" can be derived as follows.

```
E2(x, z) :- E(x, y), E(y, z);
E2(x, y) :- E(x, y);
```

This simple example demonstrates an important aspect of defining graph transformations with logic rules. In native graph transformation languages users define how the graph changes, and edges not involved in the change remain. However, when defining graph transformations logically, we must include rules to *preserve* edges not involved in the transformation (line 2).

3.1 Message Passing

Our first example involves the simple passing of a message along the directed edges of a graph, as described in [28].

```
M0(0);                    # start node
# Rule 1: Message initialization.
M(x) :- M=nil, M0(x);
# Rule 2: Message passing.
M(y) :- M(x), E(x, y);
# Rule 3: Message retention.
M(x) :- M(x), ~E(x, y);
```

Fact `M0(0)` states that initially the message is at node `0`. Rule 1 initializes the message relation `M`. It states that node `x` has a message `M` if it was specified as the initial node in `M0(x)`. Condition `M=nil` makes the rule fire (trigger) only at the start of the recursive iteration. The message propagates from node `x` to its neighbors `y` through the Rule 2. Finally, Rule 3 requires the message of a node `x` to be retained if `x` has no outgoing edges. While the rule `M(x) :- M(x), ~E(x, y)` might seem tautological as it derives a fact from itself, it plays an important role in the context of the iterative semantics (see Section 2.4): each iteration replaces the previous state entirely, so without Rule 3 messages at nodes with no outgoing edges would disappear on the next iteration.

3.2 Computing Distances in a Graph

Logica has native support for aggregation. Predicate level aggregation is computed over the specified values of the fields of the defined predicate. The following Logica program computes the minimum distance `D(x)` of a node `x` from a given `Start` node.

```
# Rule 1: Distance from the Start node is 0.
D(Start()) Min= 0;
# Rule 2: Triangle inequality.
D(y) Min= D(x) + 1 :- E(x,y);
```

`Start()` is used as a *functional predicate*: as discussed in Section 2 Logica relations can be turned to functions by having an additional "special attribute" (named `logica_value`) to store and access a relation's functional value, i.e., its value when used syntactically as a function. Here, `Start()` simply returns (a predefined) starting value. Rule 1 assigns the distance `0` to the starting node, where `D(x)` is also a functional predicate that returns the (minimum) distance of a node `x`. Rule 2 computes the distance to a node `y` as *at most* one more than the distance to its predecessor `x`. The `Min=` aggregation operator specifies that the minimum distance will be taken among all possible paths.

3.3 Solving Win-Move Games

Win-Move is a two-player combinatorial game played on a finite directed graph (representing a board) where nodes represent positions and edges represent moves [42, 15]. After choosing a starting position (node), the two players take turns moving a game pebble along the edges of the graph. A player who cannot move loses the game. We propose to compute the solution to Win-Move games in a new way, inspired by graph transformations. A concise formulation of the game is given by the logic rule:

```
Win(x) :- Move(x,y), ~Win(y).
```

A solution to the game can be computed using the 3-valued well-founded semantics [45]. The moves of the game are given by the predicate `Move`. A game starting at node `x` is objectively won if there exists a (*winning*) move to some `y` that is lost. Then, no matter how an opponent (Player II) moves from `y` (assuming there is an outgoing move to begin with), Player I can always force a win after the opponent has moved. The well-founded model assigns `Win(x)` the value *true*, *false*, and *undefined* if and only if `x` is objectively *won*, *lost*, and *drawn*, respectively. In contrast, the 2-valued *stable model* semantics does not compute the desired game solution: There can be 0, 1, or more stable models, none of which may agree with the correct, well-founded solution. Similarly, in Logica, the above rule cannot be used as-is.

Looking at the problem through a graph-transformation lens, however, we can define a new predicate `W(x,y)`, indicating that the move from `x` to `y` is a winning move. Computation of `W`, from a graph transformation perspective, amounts to selecting a collection of winning moves. In Logica this can be specified with a single rule:

```
W(x,y) :- Move(x,y), (Move(y,z1) => W(z1,z2));
```

The rule says: A move from `x` to `y` is *winning* iff for *every* opponent move from `y` to some `z1` there *exists* a move from `z1` to `z2` for Player I that is again winning! The logical implication `Move(y,z1) => W(z1,z2)` is a shorthand for the nested negation `~(Move(y,z1), ~W(z1,z2))`.

The position values can now be derived from the winning moves `W`. The source and target nodes `x` and `y` of a winning move are won and lost positions, respectively. Positions that are neither won nor lost are drawn:

```
Won(x) :- W(x, y);
Lost(y)  :- W(x,y);
Drawn(x) :- Position(x), ~Won(x), ~Lost(x);
```

Here, the unary predicate `Position` is defined as the union of the domain and range of the `Move` relation:

```
Position(x) distinct :- x in [a,b], Move(a,b);
```

3.4 Finding Paths in Evolving Graphs

Graphs that change over time [25, 8] can be naturally modeled and computed over in Logica, where time can be represented using extra positional arguments, named arguments, or as functional values. As a simple example, assume a graph that changes over time is represented by the relation `E(x,y,t0,t1)` stating that there exists an edge from node `x` to `y` added to the graph between time moments `t0` and `t1`. For simplicity we assume that time starts at moment 0, edges are transitioned instantly, and the start node is given as `Start()`. The following Logica program computes the earliest possible arrival time for each node.

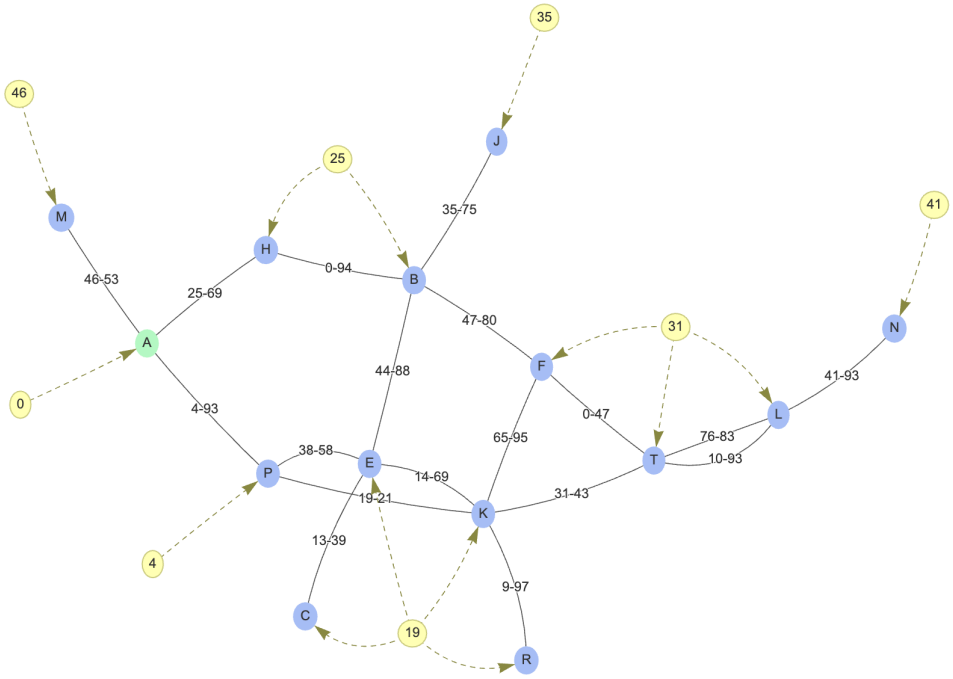

Figure 3 Visualizing the solution of pathfinding in a dynamic graph. The time of existence is shown as labels on edges; the earliest possible time of arrival is shown via yellow nodes. Section 3.6 shows how such visualizations can be created in Logica.

```
# Rule 1: Starting condition.
Arrival(Start()) Min= 0;
# Rule 2: Traversal of an edge when edge exists.
Arrival(y) Min= Greatest(Arrival(x),t0) :-
  E(x,y,t0,t1), Arrival(x) <= t1;
```

Rule 1 sets the arrival time of the start node to 0 (the assumed initial time). Rule 2 considers edge transitions: if we arrive at the source `x` of an edge before the edge expires (given by `t1`) then the arrival time of the edge's target `y` is set to the larger (latest) of `x`'s arrival time and the time `t0` that the edge was added to the graph. Here `Greatest` is a built-in function that returns the larger of its arguments (analogous to SQL's `GREATEST`), as opposed to aggregation operators like `Max=` which aggregate over multiple rows. Once the arrival times are computed, additional rules can be easily added to select specific (time aware) paths of the graph. Figure 3 gives an example of path finding over an evolving graph. An initial graph (with nodes A, B, etc., and colored blue) is given with edges labeled by their start and end times. The start node A is shown in green. The computed arrival times are displayed as additional nodes (in yellow). The graph in Figure 3 was created within Logica as further described below in Section 3.6.

3.5 Transitive Reduction of DAGs

The *transitive reduction* [3] of a directed graph is a graph with the fewest possible edges that has the same reachability as the original graph. For directed acyclic graphs (DAGs), the transitive reduction coincides with finding the (unique) *minimum equivalent subgraph*, i.e., a proper subgraph

with the fewest possible edges needed to maintain the same reachability relation as the original graph. While finding a minimum equivalent subgraph for (arbitrary) graphs with cycles is NP-complete, for DAGs, the problem can be solved in PTIME. The following Logica program computes the transitive reduction `TR(x,y)` of a DAG given by the edge relation `E(x,y)`. The program first computes the transitive closure and then uses the transitive closure to remove redundant edges.

```
# Rule 1: Transitive closure base case.
TC(x,y) distinct :- E(x,y);
# Rule 2: Transitive closure inductive step.
TC(x,y) distinct :- TC(x,z), TC(z,y);
# Rule 3: Transitive reduction.
TR(x,y) :- E(x,y), ~(E(x,z), TC(z,y));
```

Rules 1 and 2 compute the transitive closure. Rule 3 identifies edges `E(x,y)` in the original graph that cannot be bypassed by going to some other node and taking a transitive path from there. These edges are the essential edges needed to maintain reachability and make up the edges of the resulting transitive reduction graph `TR`.

3.6 Rendering Graphs with Logica

One of the benefits of Logica is the convenience of rendering graphs directly from predicate definitions. With just a few lines of Logica code and a minimal Python wrapper, it is possible to generate visually appealing and informative graph representations that highlight case-specific attributes. For example, consider the transitive reduction computed in the previous section. Instead of exporting the `TR` and `E` predicates and configuring their rendering in a separate graphing library, Logica makes it possible to define a predicate that *directly* specifies the visual attributes of the graph. The following relations define graph visualization properties in Logica highlighting the edges in both the original graph and its transitive reduction:

```
R(x, y,
  arrows: "to",
  color? Max= "rgba(40, 40, 40, 0.5)",
  dashes? Min= true,
  width? Max= 2,
  physics? Max= false,
  smooth? Max= false) distinct :- E(x, y);
R(x, y,
  arrows: "to",
  color? Max= "rgba(90, 30, 30, 1.0)",
  dashes? Min= false,
  width? Max= 4,
  physics? Max= true,
  smooth? Max= true) distinct :- TR(x, y);
```

The `R(x,y,...)` relation defines the edges of the graph and their visual properties. The first rule draws the original edges from `E(x, y)` in a light gray, dashed style with a thin line. The second rule draws the transitive reduction edges from `TR(x, y)` in bold red, solid lines. The `distinct` keyword triggers aggregation, so that each edge occurs only once and values of attributes such as `color`, `dashes`, etc., are chosen based on whether this edge belongs to the transitive reduction. The visual attributes, e.g., `arrows`, `color`, `dashes`, `width`, `physics`, and `smooth`, are directly embedded within the Logica rules. The question mark in "`color? Max=`" indicates that the `color` can have different values from different rules. The `Max=` indicates that if there are multiple rules that apply to the same edge, the maximum value will be used. The actual rendering is then handled with a Python function call:

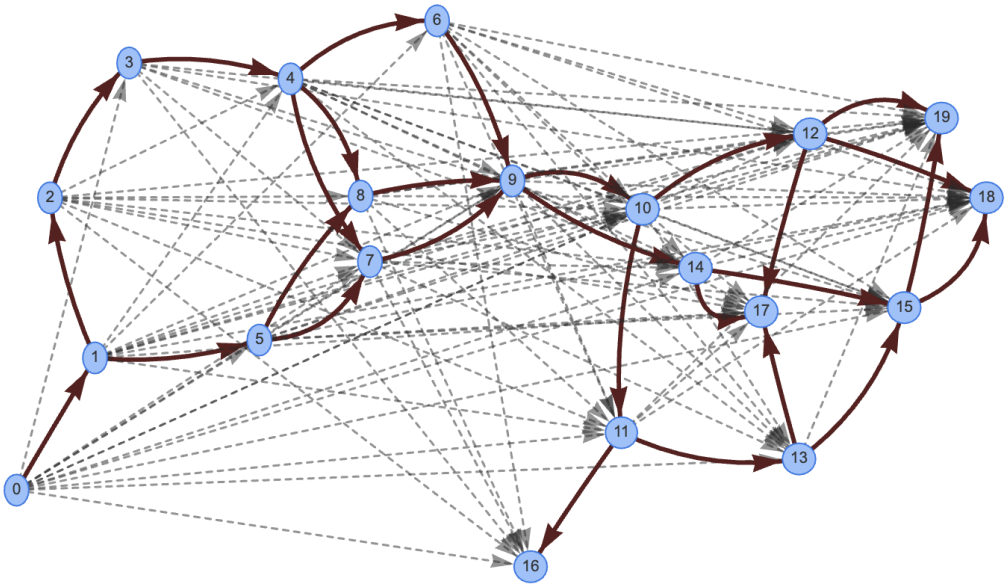

Figure 4 Visualizing a transitive reduction with Logica: Original edges are displayed with dashed lines, while edges present in the transitive reduction are highlighted with solid lines. This visualization, generated directly from Logica predicates, shows how the transitive reduction simplifies the graph while preserving reachability.

```
# from logica.common import graph
graph.SimpleGraph(
  R, extra_edges_columns=[
    "arrows", "physics", "dashes", "smooth"],
  edge_color_column="color",
  edge_width_column="width")
```

This code leverages Logica's `graph` module. The `SimpleGraph` function takes the `R` predicate, specifies which columns in `R` represent edge attributes (using `extra_edges_columns`, `edge_color_column`, and `edge_width_column`), and provides overall graph options. The resulting visualization is shown in Figure 4 (where the transitive reduction graph is overlaid over the original graph). The tight integration between logical definitions and graph rendering makes Logica a powerful tool for analyzing and understanding complex relationships in graph data. The ability to define graph properties directly in Logica, rather than relying on external tools, can significantly streamline the data exploration and visualization process.

3.7 Graph Condensation

Graph condensation [43] is a technique for simplifying a graph by collapsing strongly connected components (SCCs) into single nodes. This can be useful for visualizing the overall structure of a graph and for performing analyses that are less sensitive to the details within SCCs. In the condensed graph, each node represents an SCC, and an edge exists between two nodes if there is an edge in the original graph between nodes in the corresponding SCCs.

The following Logica program computes the condensation of a graph given by edges defined by the `E(x,y)` relation, and where all nodes are defined by a `Node(x)` predicate. We assume that transitive closure is already computed as in Subsection 3.5.

```
# Minimal node ID of the component
# .. is used as the component ID.
CC(x) Min= x :- Node(x);
CC(x) Min= y :- TC(x,y), TC(y,x);
# Compute condensation graph edges.
ECC(CC(x),CC(y)) distinct :-
  E(x,y), CC(x)!=CC(y);
```

Once the condensation is computed, it can be easily rendered using Logica's graph visualization capabilities. To enhance readability, we use the following naming conventions: nodes in the original graph are named directly with their IDs (e.g., "1", "2", "3"), while the condensed components are prefixed with "c-" (e.g., "c-1", "c-2"). The predicate definitions for rendering both the original and condensed graphs simultaneously are as follows:

```
NodeName(x) = ToString(ToInt64(x));
CompName(x) = "c-" ++ ToString(ToInt64(x));

# Edges of the original graph.
Render(NodeName(a), NodeName(b),
       physics: true, arrows: "to",
       dashes: false, smooth: true,
       color: "#33e") :- E(a, b);
# Edges of the condensation.
Render(CompName(x), CompName(y),
       physics: true, arrows: "to",
       dashes: false, smooth: true,
       color: "#33e") :- ECC(x, y);
# Mapping from the graph to
# the condensation.
Render(NodeName(ToInt64(a)), CompName(CC(a)),
       physics: false, arrows: "to",
       dashes: true, smooth: false,
       color: "#888");
```

The `NodeName` and `CompName` functions convert node and component IDs to strings with appropriate prefixes. The `Render` predicate then defines the appearance of the graph:

- Edges in the original graph (defined by `E`) are rendered as solid blue lines.
- Edges between components (defined by `ECC`) are also rendered as solid blue lines.
- Dashed gray lines connect each original node to its corresponding component. The physics is disabled between nodes and their condensation components to help with readability.

Figure 5 shows the result of using the above Logica code and the appropriate Python wrapper as described in Subsection 3.6 to view the original and corresponding condensed graph. By combining the power of Logica for defining graph structures and relationships with its seamless graph rendering capabilities, it is possible to gain valuable insights into complex systems through clear and concise visualizations.

3.8 Inferring a Taxonomic Tree

Another application of Logica lies in its ability to infer relationships from large datasets and represent them as graphs. Consider the problem of constructing a simplified taxonomic tree for a set of species, showing their evolutionary relationships leading back to common ancestors. Logica can perform this task on massive amounts of data. Here, we aim to build a taxonomic tree for *Homo sapiens* (humans), *Crocodylidae* (crocodiles), *Tyrannosaurus* (T-Rex), and *Columbidae* (pigeons). The input is a knowledge graph of triples `T(a,b,c)` and a functional predicate `L(a)` = ℓ that maps objects a to human readable labels ℓ. We define `SuperTaxon`(*item*,*parent*) as the result

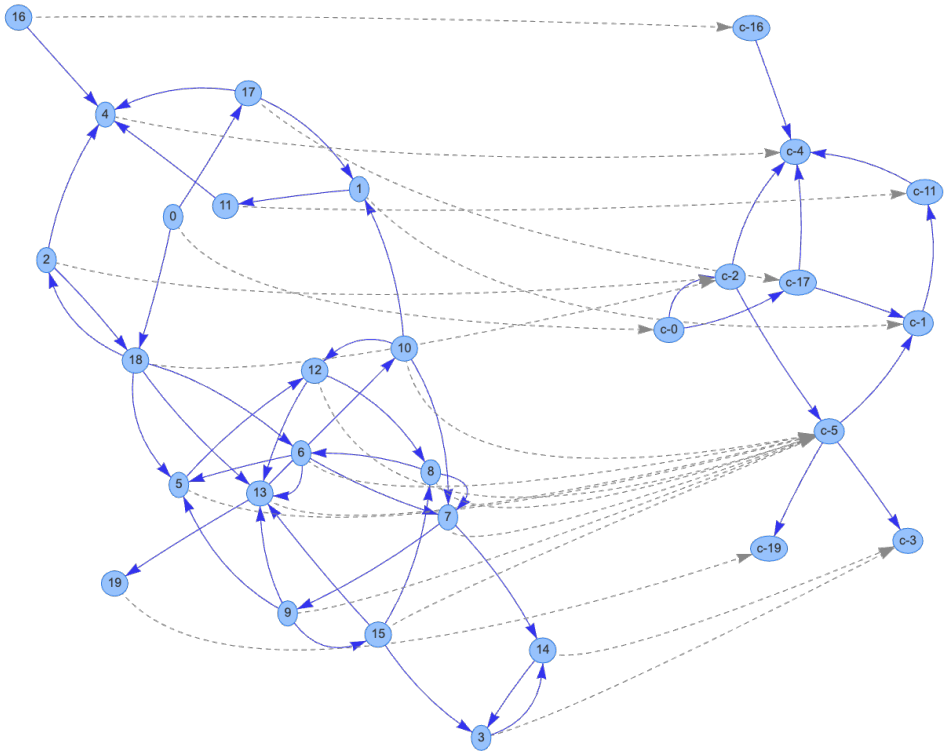

Figure 5 Graph Condensation visualized with Logica. This figure displays both the original graph and its condensed representation. Solid blue lines represent edges within the original graph and between condensed components. Dashed gray lines connect each node to its corresponding condensed component, illustrating how strongly connected components are collapsed into single nodes. This condensation simplifies the graph while preserving high-level connectivity information.

of the query `T`(*item*,`"P171"`,*parent*), which states that *parent* is a direct supertaxon of *item*, and `TaxonLabel`(x) = `L`(x) to give labels restricted to taxons. The following Logica program builds the taxonomic tree.

```
# Use unbounded depth (-1) to find ancestors.
@Recursive(E, -1, stop: FoundCommonAncestor);
E(x, item,
  TaxonLabel(x), TaxonLabel(item)) distinct :-
  SuperTaxon(item,x),
  ItemOfInterest(item) | E(item);
NumRoots() += 1 :- E(x,y), ~E(z,x);
# Stop when common ancestor is found.
FoundCommonAncestor() :- NumRoots() = 1;
```

Figure 6 shows an example output from running the program, where the GraphViz[1] library is used to render the resulting tree. The input for the example consists of the set of statements obtained by a dump of Wikidata, which contains 806M facts defined over 89M objects. When stored using DuckDB, the entire input is 13GB in size. The full recursive search was run on a Google Cloud `c2d-standard-32` instance in under 7 seconds via DuckDB with no custom index setup. The majority of the execution time was spent selecting the taxonomy edges from all possible

[1] `https://graphviz.org/`

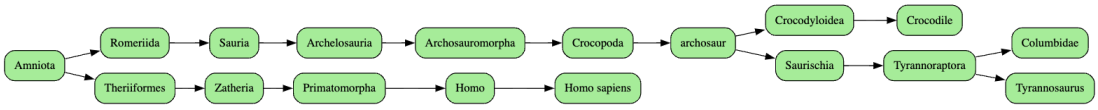

Figure 6 Inferred Taxonomic Tree for humans, crocodiles, T-Rex, and pigeons, generated by a Logica program from a large Wikidata dump. Edges represent evolutionary relationships, leading from an ancestral species to descendant species. This demonstrates that Logica can operate on very large, real-world knowledge graphs.

relations in Wikidata. Note that the number of taxons returned by the original program is large. The result shown in Figure 6 is only a sample of the obtained taxonomic tree (where the sampling is also performed by Logica; not shown above).

By expressing the problem in Logica, we leverage its ability to efficiently perform (recursive) query processing (e.g., through its compilation to DuckDB). In this case, Logica is used to quickly navigate the complex relationships within a large taxonomic database and generate output that can then be displayed visually through its Python integration (here, using the GraphViz library).

4 Performance

Table 1 Benchmark Comparison: Logica vs. Soufflé vs. Nemo vs. DuckPGQ (SQL/PGQ-Cypher).

Problem	Dataset	Nodes	Output	Runtime (s)			
				Logica	Soufflé	Nemo	DuckPGQ
Transitive Closure	G1k	1,000	1M	1.88	0.58	3.35	**0.35**
	G2k	2,000	4M	3.62	**1.73**	14.87	2.19
	G3k	3,000	9M	7.23	**3.61**	36.40	7.66
	G4k	4,000	16M	12.56	**6.98**	70.53	18.23
	G5k	5,000	25M	19.56	**10.96**	116.35	33.11
Pairwise Distances (Aggregation)	G1k	1,000	1M	**2.10**	N/A	N/A	14.46
	G2k	2,000	4M	**4.34**	N/A	N/A	116.94
	G3k	3,000	9M	**8.61**	N/A	N/A	> 200.00
	G4k	4,000	16M	**15.26**	N/A	N/A	> 200.00
	G5k	5,000	25M	**23.64**	N/A	N/A	> 200.00
Same Generation (Standard Recursion)	Tree7	180	18k	0.79	0.24	**0.02**	N/A
	Tree8	471	107k	0.92	0.29	**0.11**	N/A
	Tree9	1,205	672k	1.61	**0.62**	0.99	N/A
	Tree10	3,080	4.3M	**2.18**	3.94	5.93	N/A
	Tree11	7,781	26M	**7.71**	26.25	38.10	N/A
	Tree12	19,504	162M	**37.75**	153.46	> 200.00	N/A
Same Generation (via Aggregation)	Tree7	180	18k	1.03	N/A	N/A	**0.05**
	Tree8	471	107k	1.13	N/A	N/A	**0.17**
	Tree9	1,205	672k	1.05	N/A	N/A	**0.50**
	Tree10	3,080	4.3M	**1.20**	N/A	N/A	5.68
	Tree11	7,781	26M	**1.43**	N/A	N/A	56.67
	Tree12	19,504	162M	**2.55**	N/A	N/A	> 200.00

To help assess the performance of Logica's approach of compiling recursive queries to iterated SQL, we conducted a series of benchmarks comparing Logica against Soufflé [24], a high-performance Datalog engine written in C++, Nemo [21], a rule engine implemented in Rust, and DuckPGQ[2], an experimental SQL Property Graph Query (PGQ) implementation and extension for DuckDB.

Logica delegates query execution to an SQL engine, so its performance stems from the capabilities of that underlying engine. The benchmarks in this paper use DuckDB, a highly efficient SQL database that makes full use of modern multi-core hardware for data-parallel queries. Logica's own contribution is its declarative logical syntax, clarity of which we aimed to illustrate extensively by the examples in Sections 2 and 3.

4.1 Experimental Setup

All experiments were performed using a Google Cloud `c2d-standard-32` instance equipped with an AMD EPYC 7B13 (Milan) processor featuring 32 virtual CPUs (16 physical cores with 2 threads per core). The machine was configured with a single NUMA region and operates in 64-bit mode under the x86_64 architecture. Each core runs at a base clock speed of 2.45 GHz, with a maximum turbo frequency of 3.5 GHz. The instance is provisioned with 128 GB of RAM.

The experimental evaluations utilize two types of synthetic dataset to test different aspects of recursive query evaluation:

1. **Random Graphs (G_{Xk}):** These datasets consist of randomly generated graphs ranging from one thousand to five thousand vertices having a fixed average degree of 10, where X denotes the number vertices in thousands. With an average degree of 10, these graphs form a single large strongly connected component, so the transitive closure output is near-complete at approximately N^2 pairs. Nevertheless, computing it remains a nontrivial task: the solver must discover global reachability through the local edge structure of the graph. The pairwise distances problem over the same graphs is additionally interesting, as the computed distances are varied.
2. **Random Trees ($Tree_N$):** These datasets consist of randomly generated trees of depth N (ranging from a depth N between 7 and 12). For every non-leaf layer, each node possesses between one and five children (randomly selected).

While the input graphs are modest in size (up to 5 000 nodes), the computed outputs are large, ranging from millions to over 160 million rows (see Table 1). Thus for these problems, the size of the output is a better indicator of the computational effort than the size of the input.

We note that the reported execution times for Logica include the full end-to-end pipeline: (1) parsing the Logica code; (2) compiling the parsed code to SQL; (3) sending the SQL query to the database engine; and (4) executing the resulting SQL query using the database engine. This pipeline introduces a constant startup overhead that is visible in the smaller-sized datasets. Optimizing this compilation pipeline to minimize startup latency is a subject of future work.

We generated random input data with Logica, so it was populated in the DuckDB database, which was read when benchmarking by Logica and DuckPGQ. Data was also offloaded into CSV to be consumed by Nemo and Soufflé. DuckDB storage is highly optimized so Logica would be spending less time reading data, however by the nature of these benchmarks the output is orders of magnitude larger than input and thus the total reading time is making an insignificant contribution to overall execution time.

All benchmark queries and scripts are available as Jupyter notebooks in the Logica repository.[3]

[2] `https://duckpgq.org/`
[3] `https://github.com/EvgSkv/logica/tree/main/examples/graph/tgdk`

4.2 Performance Analysis

We consider four types of computations for evaluating performance: (1) transitive closure; (2) pairwise distances (using aggregation); (3) finding nodes that are part of the same generation (using standard recursion); and (4) same generation via aggregation. The results of the experimental evaluation are summarized in Table 1 and are further discussed below for each type of computation.

4.2.1 Transitive Closure

Implementation of Transitive Closure in Logica is discussed in Section 3.5. In the benchmarks we use a linear recursion variant for efficiency:

```
@Recursive(TC, stop: Done);
TC(a, b) distinct :- G(a, b);
TC(a, c) distinct :- TC(a, b), G(b, c);
PrevTC(a, b) :- TC(a, b);
Done() :- TC(a, b) => PrevTC(a, b);
```

The second rule uses linear recursion (`TC(a, b), G(b, c)`) rather than quadratic (`TC(a, b), TC(b, c)`), which is cheaper in practice. The `stop` directive halts iteration once the fixpoint is reached, i.e., when no new pairs are produced.

For pure reachability tasks on the G_{Xk} datasets, specialized engines show an advantage. DuckPGQ and Soufflé outperform Logica on smaller graphs. However, Logica's performance remains within the same order of magnitude. As the size of the dataset grows, the compilation overhead becomes negligible in Logica relative to the total runtime, and as shown, Logica times begin to outperform DuckPGQ and move closer towards those of Soufflé.

4.2.2 All-Pairs Shortest Paths

All-pairs shortest paths problem is finding the minimum-length path between each pair of nodes, that is finding matrix of distances between all pairs of vertices. Given predicate `E` containing all edges, in Logica this problem is solved as follows.

```
D(x, y) Min= 1 :- E(x, y);
D(x, z) Min= D(x, y) + D(y, z);
```

Recursion stop condition for `D` and `SG` is done analogously to the Transitive Closure program above.

This problem highlights the differences in feature sets and scalability among the three systems. In particular, Soufflé is excluded from this benchmark because it restricts the use of aggregation within recursive cycles (stratification). In the case of DuckPGQ, we observed that the query execution time increased rapidly with dataset size, resulting in timeouts for datasets larger than G_{2k}. Logica, however, successfully compiled the logic into recursive SQL using standard arithmetic increments. By leveraging the database's native ability to handle aggregation within recursion, Logica completed the G_{5k} task in under 24 seconds, whereas the PGQ extension for DuckDB failed to complete within the 200-second time limit.

4.2.3 Same Generation

The Same Generation problem is to find all pairs of nodes at the same depth in a tree. This problem can be solved using two fundamentally different approaches, illustrating the tradeoff between compilation overhead and runtime scaling.

Standard Recursion. The first approach to computing generations uses pure recursive rules without aggregation. The following is a Logica implementation of Same Generation for the predicate `Parent`, where the implementation in Soufflé is analogous.

```
SG(x, y) :- Parent(p, x), Parent(p, y);
SG(x, y) :- SG(a, b), Parent(a, x), Parent(b, y);
```

As shown, Soufflé dominates on small trees where Logica pays a consistent compilation overhead. However, as the trees grow exponentially with depth, Logica outperforms Soufflé in terms of overall execution time. Soufflé requires dramatically more time on deeper trees while Logica's runtime cost grows much more gradually. Unlike Logica and Soufflé, DuckPGQ is not designed to express this type of query, as shown by N/A's across each of the corresponding benchmarks in Table 1.

Using Aggregation. The second approach for computing the Same Generation problem uses aggregation to first find the depth of each node in the tree. The implementation of this approach in Logica is shown below for input predicates `Parent` and `Node`.

```
Gen(x) Min= x :- Node(x);
NextGen(Gen(x)) Min= Gen(y) :- Parent(x, y);
Gen(y) Min= NextGen(Gen(x)) :- Parent(x, y);
SG(x, y) :- Gen(x) = Gen(y);
```

This implementation defines functional predicates `Gen` and `NextGen`. The predicate `Gen` maps each node to the representative of its generation, i.e., the smallest node in the generation. The predicate `NextGen` connects the generation of parent nodes to the generation of their children. These two predicates are defined through mutual recursion using `Min` aggregations such that the generation that a child belongs to is the next generation after the generation of its parent.

To answer the query using aggregation in DuckPGQ, we assume the root of the tree is known, and compute same generations using distances from nodes to the root. However, as in the pairwise-distances problem, Soufflé cannot be used to compute generations since it restricts the use of aggregation through recursive cycles. As shown in Table 1, DuckPGQ performs well on shallow trees, but as tree depth increases, it scales substantially slower than Logica, eventually timing out. Logica's performance remains stable across all tree sizes. Once Logica's execution time is no longer dominated by compilation overhead in the examples, the aggregation-based approach implemented in Logica scales efficiently even on the deeper trees in the benchmark.

4.3 Discussion

The evaluation results highlight the benefits of compiling logic programs to SQL and delegating execution to a modern analytical database engine. DuckDB, which serves as Logica's backend in these benchmarks, is designed for efficient parallel execution on a single machine. As the output grows, DuckDB effectively utilizes all available cores – for instance, in the Same Generation problem, Logica computes 26 million output rows (Tree11) in 7.71 seconds, while computing 162 million rows (Tree12) takes 37.75 seconds. The output grew 6× while execution time grew under 5×, demonstrating sub-linear scaling relative to output size. By delegating execution to DuckDB, Logica achieves performance competitive with specialized engines like Soufflé and DuckPGQ.

5 State of the Art

Our work is related to three approaches for graph transformation: rule-based, logic-based, and graph query languages. This section reviews the major systems and further examines how these approaches handle challenges related to expressiveness, scalability, and temporal reasoning, compared to Logica.

5.1 Rule-Based Graph Transformation Systems

Classical graph transformation systems have developed consistently since the 1970s, when rule-based graph transformation emerged as a research field [10].

The Attributed Graph Grammar system (AGG) [12] represents one of the earliest implementations of the algebraic single-pushout approach [11]. AGG supports the visual editing of graphs and rules, enabling users to define transformations in a visual programming environment. The GRaph-based Object-Oriented VErification (GROOVE) system [18] emphasizes efficient exploration of state spaces comprising generated graphs by applying various search strategies (.eg., depth-first), with built-in model checking and isomorphism recognition. GrGen.NET [22] utilizes generative programming to compile graph rewrite rules into highly optimized .NET code for faster execution speed, proven by the Varró benchmark [46].

However, because these rule-based graph transformation systems prioritize formal verification and model transformation capabilities over computational efficiency, they face fundamental scalability barriers when applied to large-scale graphs. For instance, GROOVE suffers from dramatic growth in state space size as problem complexity increases. Furthermore, these tools function as standalone applications, making it difficult to integrate them with modern data processing pipelines.

5.2 Logic-Based Graph Transformation

Logic programming systems also adopt a rule-based approach for graph transformation. By leveraging database optimization techniques, Datalog-based systems show particular promise for scalable graph analytics.

In Section 4 we compared Logica to Soufflé [24] and Nemo [21]. Soufflé compiles Datalog programs to optimized parallel C++ (our benchmarks use its compiled mode), achieving high performance on tasks like transitive closure and reachability, although it does not support aggregation within recursive cycles. Nemo is a recent rule engine implemented in Rust, designed for large-scale reasoning with knowledge graphs and ontologies, capable of ingesting data from various sources including CSV, RDF, and SPARQL endpoints. At present Nemo's evaluation is single-threaded, which places it at a disadvantage on our multi-core benchmark hardware; nevertheless its Rust implementation is very promising, and once multi-threaded evaluation is added Nemo may well outperform Logica on the problems we study. RecStep [13] achieved performance improvements on graph transformation tasks by building Datalog engines on top of parallel relational database systems. DcDatalog [48] leverages shared-memory multicore architectures to optimize graph query execution time.

Several other Datalog-based systems are relevant to graph processing. RDFox [32] is a high-performance in-memory RDF store with built-in Datalog reasoning, supporting parallel incremental materialization over large knowledge graphs. SociaLite [37] extends Datalog with recursive aggregates for efficient graph analytics. Vadalog [4] targets knowledge graph reasoning with Warded Datalog$^{\pm}$. Flix [29] extends Datalog with user-defined lattices and first-class functions for expressing fixed-point computations.

The declarative nature of these approaches shares conceptual foundations with schema mapping systems, which has been widely used in data integration. Modern schema mapping systems like LogMap, an actively maintained scalable ontology matching system with built-in reasoning engine [23], and CODI [20], which introduces probabilistic-logical alignment into data integration. Most of these systems scale from small to medium scale of graphs, and face challenges with large-scale graph processing.

Logica [41] distinguishes itself by compiling logical queries into existing database systems such as DuckDB, and Google BigQuery. This approach leverages established database optimization techniques for graph transformation while benefiting from continuous database improvements and enabling large-scale graph processing.

5.3 Graph Query Languages

Besides logic programming languages for graph processing, there are other graph query languages, which usually have close relationships to graph database systems, for instance, Cypher [16] to Neo4j [30] and DuckPGQ (which performance is also shown in Section 4). These query languages are usually designed for different query needs.

Cypher [16] provides a visual graph matching mechanism for Neo4j, enabling users to express graph patterns using an intuitive syntax while supporting grouping and aggregation. Gremlin [34], part of Apache TinkerPop project, is a functional programming language natively support both imperative and declarative querying for graph traversal. SPARQL [33] serves for querying RDF knowledge graphs. Its ability to federate queries across multiple data sources makes it particularly suitable for semantic web applications.

Recent work by Bonifati et al. [6, 7, 5] addresses declarative graph-to-graph transformations within the Cypher ecosystem, extending GQL/SQL-PGQ foundations with rule-based transformation specifications. While their approach enables property graph transformations directly within graph database backends, Logica takes a complementary path by compiling to SQL, leveraging established database engines for scalability while maintaining full Datalog expressiveness for recursive graph computations.

6 Conclusion

We have presented a novel approach to implementing graph transformations using Logica, a free and open-source logic programming language designed for parallel databases. While Logica is typically used for data processing and analysis, we have shown that it can also effectively be used for graph transformations. By leveraging Logica's ability to process large datasets in a parallel and efficient manner, graph transformations can be performed efficiently. By showcasing several examples, including the Win-Move problem, dynamic path finding, transitive reduction, graph condensation, and graph querying, we aimed to demonstrate that graph transformation problems can be expressed in an intuitive and efficient manner. The examples show how describing graph transformations using logic rules can lead to concise declarative specifications that also improve readability. For example, our approach models time-varying graphs in a principled manner, an issue that can be challenging in classical graph transformation languages.

In future work, we plan to explore more complex graph transformation patterns, including rewritings that may require solving NP-hard problems. We also plan to benchmark our approach on large complex reasoning problems, which would be valuable for deeper understanding of Logica's performance on real-world datasets.

We hope that Logica, with its ability to leverage underlying SQL engines, can make graph transformations a more widely accessible and practical approach for managing complex and dynamic data.

References

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

2 Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci.*, 43(1):62–124, August 1991. doi:10.1016/0022-0000(91)90032-Z.

3 A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972. doi:10.1137/0201008.

4 Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *VLDB*, 11(9):975–987, 2018. doi:10.14778/3213880.3213888.

5 Angela Bonifati. Versatile property graph transformations. *Proc. VLDB Endow.*, 18(12):5516–5526, August 2025. doi:10.14778/3750601.3760517.

6 Angela Bonifati, Filip Murlak, and Yann Ramusat. Transforming property graphs. *Proc. VLDB Endow.*, 17(11):2906–2918, July 2024. doi:10.14778/3681954.3681972.

7 Angela Bonifati, Yann Ramusat, Filip Murlak, Amela Fejza, and Rachid Echahed. Dtgraph: Declarative transformations of property graphs. *Proc. VLDB Endow.*, 17(12):4265–4268, August 2024. doi:10.14778/3685800.3685851.

8 Arnaud Casteigts, Anne-Sophie Himmel, Hendrik Molter, and Philipp Zschoche. Finding temporal paths under waiting time constraints. *Algorithmica*, 83(9):2754–2802, September 2021. doi:10.1007/s00453-021-00831-w.

9 Francisco de la Parra and Thomas Dean. Survey of graph rewriting applied to model transformations. In *International Conference on Model-Driven Engineering and Software Development*, pages 431–441, 2014. doi:10.5220/0004731504310441.

10 Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. Graph and model transformation. *Monographs in Theoretical Computer Science. Springer*, page 17, 2015.

11 Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation–part ii: Single pushout approach and comparison with double pushout approach. In *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations*, pages 247–312. World Scientific, 1997.

12 Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The agg approach: Language and environment. In *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999.

13 Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M Patel. Scaling-up in-memory datalog processing: Observations and techniques. *Proceedings of the VLDB Endowment*, 12(6), 2019. doi:10.14778/3311880.3311886.

14 Maribel Fernández, Hélène Kirchner, Bruno Pinaud, and Jason Vallet. Labelled graph rewriting meets social networks. In *International Workshop on Rewriting Logic and Its Applications*, volume 9942 of *LNCS*, pages 1–25. Springer, 2016. doi:10.1007/978-3-319-44802-2_1.

15 Jörg Flum, Max Kubierschky, and Bertram Ludäscher. Total and partial well-founded datalog coincide. In *Intl. Conf. on Database Theory (ICDT)*, LNCS 1186, pages 113–124. Springer, 1997. doi:10.1007/3-540-62222-5_40.

16 Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018. doi:10.1145/3183713.3190657.

17 Michael Gelfond. Answer sets. In Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 285–316. Elsevier, 2008. doi:10.1016/S1574-6526(07)03007-6.

18 Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using groove. *International journal on software tools for technology transfer*, 14(1):15–40, 2012. doi:10.1007/S10009-011-0186-X.

19 Tomasz Gogacz and Jerzy Marcinkowski. All-instances termination of chase is undecidable. In *Proc. ICALP*, volume 8573 of *LNCS*, pages 293–304. Springer, 2014. doi:10.1007/978-3-662-43951-7_25.

20 Jakob Huber, Timo Sztyler, Jan Noessner, and Christian Meilicke. Codi: Combinatorial optimization for data integration–results for oaei 2011. *Ontology Matching*, 134, 2011.

21 Alex Ivliev, Stefan Ellmauthaler, Lukas Gerlach, Maximilian Marx, Matthias Meißner, Simon Meusel, and Markus Krötzsch. Nemo: First glimpse of a new rule engine. In *Proceedings of the 39th International Conference on Logic Programming (ICLP 2023)*, 2023.

22 Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. Grgen. net: The expressive, convenient and fast graph rewrite system. *International Journal on Software Tools for Technology Transfer*, 12(3):263–271, 2010. doi:10.1007/S10009-010-0148-8.

23 Ernesto Jiménez-Ruiz and Bernardo Cuenca Grau. Logmap: Logic-based and scalable ontology matching. In *International Semantic Web Conference*, pages 273–288. Springer, 2011. doi:10.1007/978-3-642-25073-6_18.

24 Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, LNCS, pages 422–430. Springer, 2016. doi:10.1007/978-3-319-41540-6_23.

25 David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *J. Comput. Syst. Sci.*, 64(4):820–842, June 2002. doi:10.1006/jcss.2002.1829.

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

27 Tim Kräuter, Adrian Rutle, Harald König, and Yngve Lamo. Formalization and analysis of bpmn using graph transformation systems. In *International Conference on Graph Transformation*, pages 204–222, 2023. doi:10.1007/978-3-031-36709-0_11.

28 Barbara König. Graph transformation meets logic. GReTA - Graph Transformation Theory and Applications Symposium, 2020. URL: https://www.irif.fr/~greta/event/nov20th2020-knig/.

29 Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From Datalog to Flix: A declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, 2016.

30 Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, volume 2324 (36), pages 141–147, 2013.

31 Manfred Nagl. Graph rewriting systems and their application in biology. In *Mathematical Models in Medicine*, pages 135–156. Springer, 1976.

32 Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A highly-scalable RDF store. In *The Semantic Web – ISWC 2015*, Lecture Notes in Computer Science. Springer, 2015. doi:10.1007/978-3-319-25010-6_1.

33 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009. doi:10.1145/1567274.1567278.

34 Marko A Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th symposium on database programming languages*, pages 1–10, 2015. doi:10.1145/2815072.2815073.

35 Francesc Rosselló and Gabriel Valiente. *Graph Transformation in Molecular Biology*, pages 116–133. Springer, 2005. doi:10.1007/978-3-540-31847-7_7.

36 Francesc Rosselló and Gabriel Valiente. Chemical graphs, chemical reaction graphs, and chemical graph transformation. *Electronic Notes in Theoretical Computer Science*, 127(1):157–166, 2005. Proceedings of the International Workshop on Graph-Based Tools. URL: https://www.sciencedirect.com/science/article/pii/S157106610500112X.

37 Jiwon Seo, Stephen Guo, and Monica S. Lam. SociaLite: Datalog extensions for efficient social network analysis. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE 2013)*, pages 278–289, 2013. doi:10.1109/ICDE.2013.6544832.

38 Evgeny Skvortsov. Logica project, 2023. URL: https://logica.dev/.

39 Evgeny Skvortsov, Yilin Xia, Bertram Ludäscher, and Shawn Bowers. Logica tgdk: Graph transformation, December 2025. URL: https://tinyurl.com/LogicaGraphTransformation.

40 Evgeny S. Skvortsov, Yilin Xia, Shawn Bowers, and Bertram Ludäscher. The logica system: Elevating SQL databases to declarative data science engines. In *International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog-2.0)*, volume 3801 of *CEUR Workshop Proceedings*, pages 69–73, 2024. URL: https://ceur-ws.org/Vol-3801/short5.pdf.

41 Evgeny S. Skvortsov, Yilin Xia, and Bertram Ludäscher. Logica: Declarative data science for mere mortals. In *International Conference on Extending Database Technology*, pages 842–845, 2024. doi:10.48786/edbt.2024.84.

42 Cedric A.B. Smith. Graphs and composite games. *Journal of Combinatorial Theory*, 1(1):51–81, June 1966.

43 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

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

45 Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The Well-founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):619–649, July 1991. doi:10.1145/116825.116838.

46 Gergely Varró, Andy Schurr, and Dániel Varró. Benchmarking for graph transformation. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 79–88. IEEE, 2005.

47 Victor Vianu. Rule-based languages. *Annals of Mathematics and Artificial Intelligence*, 19(1):215–259, March 1997. doi:10.1023/A:1018907806177.

48 Jiacheng Wu, Jin Wang, and Carlo Zaniolo. Optimizing parallel recursive datalog evaluation on multicore machines. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1433–1446, 2022. doi:10.1145/3514221.3517853.