The Logica System: Elevating SQL Databases to Declarative Data Science Engines

Evgeny Skvortsov[1,*], Yilin Xia[2,†], Shawn Bowers[3,†] and Bertram Ludäscher[2,†]

[1]Google LLC, WA, USA
[2]University of Illinois Urbana-Champaign, School of Information Sciences, IL, USA
[3]Gonzaga University, Department of Computer Science, Spokane, WA, USA

Abstract

Logica (= Logic + aggregation) is a freely available, open-source, feature-enhanced version of Datalog that automatically compiles logic rules to a number of popular SQL platforms (DuckDB, SQLite, PostgreSQL, and BigQuery). Logica combines beginner-friendly declarative features of Datalog with advanced analytical features needed by data science and ML practitioners when processing real-world data. Since Logica is built on top of mature SQL implementations, these features can be executed robustly and scalably. Logica allows beginners to seamlessly progress from simple textbook examples to intermediate and advanced use cases. We introduce Logica with examples that combine aggregation, recursion, and negation in interesting and powerful ways. Additional advanced examples (maximum flow, matrix inversion, etc.) are demonstrated in an online notebook. Logica source programs are compiled into (a) self-contained SQL scripts (for non-recursive and shallow-recursive problems) or (b) Python-driven iterations of SQL queries (when deep recursion is needed). Logica's practical and theoretical expressive power thus extends both SQL and (pure) Datalog. The Logica system has been used for data science applications and training in industry, and in graduate-level courses in academia.

Keywords

Datalog-to-SQL compilation, aggregation, non-stratified negation, declarative data science applications

1. Introduction

Datalog has a long history in databases [1], e.g., for studying the expressive power of queries [2], as a logical foundation for nonmonotonic reasoning [3, 4], and as a language for teaching the foundations of databases [5]. The resurgence of Datalog in academia and industry [6] yielded new application areas that bridge the gap between specification and implementation: e.g., in program analysis [7], declarative networking [8, 9], knowledge graphs [10], and ML/AI [11, 12]. For these and other use cases, a number of specialized Datalog systems and prototypes have been implemented, e.g., see [13, 7, 10] among others.

Somewhat surprisingly, however, despite these successes there has been a lack of freely available, scalable implementations of Datalog that support real-world data-science applications. In contrast, the ubiquity and success of Python can be explained by (i) the fact that it is freely and widely available, and (ii) it is a language and system that can grow with the experience and needs of learners and users: with Python there is a continuous path—from beginner to expert—within a single framework. If the user-friendly, declarative features of Datalog could be combined with the robust, well-engineered features of SQL databases in a widely available implementation, a similar path could allow beginners to advance from simple, declarative queries to more complex data-intensive analysis use cases.

We begin to address this challenge using Logica, a freely available, open-source Datalog variant, designed to combine the declarative features of a logical rule language with features needed in real-world data science applications. Logica is a descendant of Yedalog [14] (and thus Dyna [12]) and inherits several features through this lineage, e.g., a programmer-friendly syntax for aggregators,

Datalog 2.0 2024: 5th International Workshop on the Resurgence of Datalog in Academia and Industry, October 11, 2024, Dallas, Texas, USA

*Corresponding author.
†These authors contributed equally.
evgenys@google.com (E. Skvortsov); yilinx2@illinois.edu (Y. Xia); bowers@gonzaga.edu (S. Bowers); ludaesch@illinois.edu (B. Ludäscher)

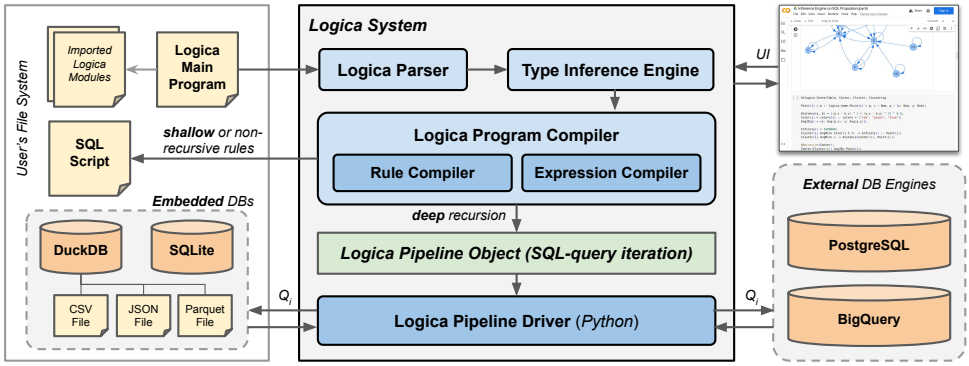

Figure 1: System Architecture: Logica supports a module system for importing libraries (top left); users write programs either locally (top left), e.g., as part of a larger Python script, or from within Jupyter notebooks (top right); programs are parsed, type checked (top middle), and, if well-formed, 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).

functional predicates, user-defined functions, and complex data types. Logica source code, tutorials, and demonstration notebooks are available online [15, 16].

In Section 2 we provide an overview of the Logica system architecture and its components. The system compiles rules into (a) self-contained SQL scripts, or (b) to Python-driven pipelines that chain together SQL queries (if deep recursion is needed). In Section 3 we introduce some of the key features of the Logica language by example. Advanced features include user-defined, rule-based aggregation, support for positional and name-based predicates, and numeric computations (e.g., to compute the PageRank of a network). These and additional examples are available through and online demonstration [16]. In Section 4, we discuss future work.

2. The Logica System Architecture

An overview of the system architecture is depicted in Figure 1: The user can work with Logica from the command line or use an interactive UI (e.g., Jupyter notebook). The *main program* can import functions and other rules via a module system. In the main program (or Jupyter cell), the user specifies a subset of top-level predicates that should be evaluated (skipping rules that are not needed). After parsing and analyzing the main Logica file and all its dependent (imported) files, a collection of SQL queries is created, in the dialect of the target database engine. Since the different engines each have their own limitations and idiosyncrasies, information from the *type inference engine* is used to create the correct SQL code for each platform.

If the set of rules to be evaluated is non-recursive (or if the user has indicated via a *directive* that a fixed-depth, non-iterative recursion is sufficient), a self-contained SQL script is generated that can be executed en bloc, in the required order. For long-running queries the user can monitor rule execution in the UI: Top-level predicates and the predicates they depend on are rendered as they are being evaluated[1], so the user knows which (iterated) relations are complete or still running, respectively.

System Components. The *Logica Parser* (Figure 1) reads and analyzes the main program and all dependent (imported) modules. The resulting syntax tree is represented by a JSON-like object. The *Type Inference Engine* works on this object to infer the types of predicate arguments and expressions in the bodies of rules. Type inference is used to catch programming errors in Logica rules, producing high-level, user-friendly messages (instead of forcing the user to debug faulty SQL). The type-annotated syntax tree is then passed to the *Logica Program Compiler*, which generates a collection of interdependent SQL queries. The *Expression Compiler* sub-component turns Logica value expressions into equivalent SQL expressions. The *Rule Compiler* takes a single conjunctive rule and translates it into a single SQL

[1] in ASCII mode when running in a terminal, or as a dynamic Graphviz-rendered pipeline graph in Jupyter

query. If the program is non-recursive or the expected runtime recursion depth is shallow (e.g., < 100), the compiler can output a reasonably-sized, self-contained SQL script that produces a result table for each compiled predicate. For predicates requiring deep recursion, Logica will create a JSON-like pipeline object to execute the generated SQL queries via the Python-based *Logica Pipeline Driver*. The latter executes SQL queries Q_i iteratively (Figure 1), stage-by-stage until a fixpoint or a user-defined termination condition is reached. Intermediate results are stored in the target database.

3. Logica Rules by Example

In Logica syntax, predicate names are capitalized, variable names begin with lowercase letters, negation is denoted by "~", and rules are terminated with semicolons.[2] Logica programs consist of sets of *statements*, which are either rules or *directives* (e.g., imports). The following stratified Logica rules specify how to compute the lowest-common-ancestor (LCA) `a` of two nodes `x` and `y`:

```
Ancestor(x,a) :- Parent(x,a);
Ancestor(x,a) :- Ancestor(x,z), Ancestor(z,a);
CommonAncestor(x,y,a) :- Ancestor(x,a), Ancestor(y,a), x != y;
NonLCA(x,y,a) :- CommonAncestor(x,y,a), CommonAncestor(x,y,b), Ancestor(b,a);
LCA(x,y,a) :- CommonAncestor(x,y,a), ~NonLCA(x,y,a);
```

When compiling Datalog to plain SQL there are several natural approaches for handling recursion, e.g., (1) use `WITH RECURSIVE` and *Common Table Expressions* (CTEs), or (2) unroll recursive rules to a fixed depth k. Both approaches have pros and cons, e.g., (1) cannot handle non-linear recursion, while (2) may result in incomplete answers when the recursion depth required exceeds the user-defined "unroll-depth" k set for a given program. For shallow-recursive problems Logica employs (2), as this is often sufficient in practice. The double-recursive `Ancestor(x,a)` specification above, e.g., can handle graphs with diameter $d \leq 2^k$ given an unroll-depth of k.

Loans-by-Month: Named Attributes and Basic Aggregation. Logica supports the traditional *positional* Datalog syntax (used above) and the *name-based* syntax from SQL. The latter makes Logica, like SQL, more robust to schema changes[3] and is a practical necessity when dealing with real-world tables that have more than a few columns. Consider, e.g., a relational schema for `LibLoans` giving the number of `loans` (checkouts) for different `item` types in a given `month`. The following rules use both positional and named syntax to compute the number of checkouts for items that are popular (having at least 30K total loans).

```
AnyMonth() = i + 1 :- i in Range(12);
ItemCount(item) += loans :- LibLoans(item:,loans:);
PopularItem() = item :- ItemCount(item) > 30000;
LoansByMonth(item:PopularItem(),month:AnyMonth()) += 0;
LoansByMonth(item:,month:) += loans :- LibLoans(item:, month:,loans:),item = PopularItem();
```

The rule in `1` uses the built-in function `Range()` to define a multi-valued function `AnyMonth()` that returns integers $1, \ldots, 12$. All Logica relations have an additional "special attribute" `logica_value` used to store and access a relation's functional value, i.e., its value when used syntactically as a function. The rule in `2` uses *named* attributes for `LibLoans()` to sum the total number of checkouts for each item. Each named attribute (e.g., `item:` in rule `2`) implicitly defines a *variable* of the same name (here: `item`). Rule `3` finds popular items: `logica_value` is accessed via `ItemCount(item)` in the body. Rules `4` and `5` compute the number of loans per month for each of the popular items. This example uses aggregation but no recursion and thus compiles into a single (stand-alone) SQL script.

PageRank: Aggregation through Recursion. PageRank quantifies the importance of a web page p based on the importance of pages that link to it [17]. Let $M(p)$ be the set of pages that link to p; $L(p)$ be the number of links from p; and N be the total number of pages in a web graph. For a damping factor d ($0 \leq d \leq 1$), the PageRank $\mathsf{PR}(p_i)$ of page p_i is recursively defined as:

[2] Logica, like SQL, also supports bag semantics and a `distinct` keyword.
[3] For example, adding new columns to a schema usually will not break a query.

$$\mathsf{PR}(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{\mathsf{PR}(p_j)}{L(p_j)}.$$

Intuitively, a page is important if many pages or a few important pages link to it. PageRank can be interpreted as a probability distribution modeling the behavior of a web-surfer following links from a random page until becoming bored, then jumping to another random page, clicking links, etc. Thus, PR is the probability the surfer visits a page and d is the probability the surfer becomes bored and jumps to another page. These two rules implement PageRank in Logica:

```
PageRank(x) += ResetProb() / N() :- Page(x);
PageRank(y) += (1.0 - ResetProb()) * PageRank(x) / Degree(x) :- Link(x,y);
```

Here `ResetProb()` is $1-d$; `N()` is N; and `Degree(x)` is L from above. This code assumes a fixed-depth (shallow) recursion. Alternatively, consider this iterative (pipelined) version:

```
@Recursive(PageRank(), 100, iterative:true,stop:ChangeIsSmall);
PageRank(x) += ResetProb() / N() :- Page(x), MonitorIteration();
PageRank(y) += (1.0 - ResetProb()) * PageRank(x) / Degree(x) :- Link(x,y);
```

The directive `@Recursive` states that `PageRank()` rules are iterated until a stop condition becomes true or a maximum of 100 iterations is reached. The former is given by a `ChangeIsSmall()` predicate that compares the change between consecutive `PageRank()` iterations to a given ϵ-value (see [16] for details). `MonitorIteration()` is used to encapsulate the call to `ChangeIsSmall()`.

k-Means: Collection Types and Mutual Recursion. The k-means algorithm partitions a collection of rows into k clusters using a distance metric. First, k-means assigns rows randomly to k initial clusters and then computes each cluster's *centroid*. The rules then reassign each row to the nearest cluster centroid; recalculates centroids; and repeats until a fixpoint is reached. The following rules implement k-means for $k = 3$ (rows represent points with `X` and `Y` coordinates):

```
1 Distance(p1,p2) = ((p1.X - p2.X) ^ 2 + (p1.Y - p2.Y) ^ 2) ^ 0.5;
2 Color(i) = colors[i] :- colors = ["red", "green", "blue"];
3 Cluster(i) ArgMin= Color(i % 3) -> Infinity() :- Point(i);
4 Cluster(i) ArgMin= c -> Distance(Centroid(c), Point(i));
5 Avg2D(p) = {X:Avg(p.X), Y:Avg(p.Y)};
6 Centroid(Cluster(i)) Avg2D= Point(i);
```

This also demonstrates Logica's support for list and record types, used to represent the k clusters (as colors) and point objects, respectively. `Point()` is an input that maps indexes to point objects; `Distance()` computes the Euclidean distance between two points; and `Color()` assigns index values to color strings. The `Cluster()` rules use indirect recursion with the `ArgMin` aggregation function. "`ArgMin=` x `->` f" computes the argument x of f with a minimal f-value. Rule 3 assigns each point to one of the three clusters and rule 4 finds the cluster with the closest centroid. Cluster centroids are found in line 6 using the user-defined aggregate function `Avg2D` (line 5).

Win-Move: Negation through Recursion. Win-move is a two-player game played on a digraph. Nodes represent positions, edges possible moves. A play starts at a position and players take turns moving a pebble along the edges. A player who reaches a terminal node cannot move and loses. Such games can be solved via the well-founded semantics (WFS) and a single non-stratified rule [3]: `Win`(x) `:-` `Move`(x, y), $\neg$`Win`(y). Using WFS, `Win`(x) is *true*, *false*, or *undefined* iff position x is objectively *won*, *lost*, or *drawn*, respectively. Logica's production rule semantics (cf. [4]) is similar but different from WFS: Instead of detecting the drawn positions, Logica alternates their value between won and lost. The following rewritten rules compute a *reified* version of the 3-valued solution, i.e., the value of a position `x` is *explicitly* captured by 2-valued predicates `Won()`, `Lost()`, and `Drawn()`, respectively; something not readily available in the WFS.

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

The crux is the implicitly ∀-quantified variable y in rule 2's body, a construct discussed in [18] and [3], which in this case represents the first-order formula $(\forall y)(\texttt{Move}(x, y) \rightarrow \texttt{Won}(y))$. We note that in addition to the above rewriting, it is also possible (although slightly more involved) to use the single non-stratified win-move rule together with additional rules to stop iteration, similar to the PageRank example earlier, and then compute the drawn positions as a post-processing step (within the same Logica program). Another alternative approach in Logica for solving win-move games uses numerical scores and a time-delayed "reward value"; see [16].

4. Summary and Future Plans

Logica is a freely available, open-source, feature-enhanced version of Datalog for beginners and advanced users [15]. Its Datalog-to-SQL compilation makes it a practical tool for declarative data science and problem solving. Experience with training in industry and graduate-level teaching suggest that Logica can be an effective tool for learning databases, data science, and logic programming. We plan to create a public repository of teaching materials, including specialized (thematic) tutorials, and a suite of educational notebooks. We also hope to build an active Logica community that will join our efforts.

References

[1] D. Maier, K. T. Tekle, M. Kifer, D. S. Warren, Datalog: concepts, history, and outlook, in: Declarative Logic Programming: Theory, Systems, and Applications, 2018, pp. 3–100.
[2] S. Abiteboul, V. Vianu, Datalog extensions for database queries and updates, Journal of Computer and System Sciences 43 (1991) 62–124.
[3] A. Van Gelder, K. A. Ross, J. S. Schlipf, The Well-founded Semantics for General Logic Programs, Journal of the ACM 38 (1991) 619–649.
[4] V. Vianu, Datalog Unchained, in: ACM PODS, 2021, pp. 57–69.
[5] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.
[6] M. Alviano, A. Pieris (Eds.), 4th Intl. Workshop on the Resurgence of Datalog in Academia and Industry, volume 3203 of *CEUR Workshop Proceedings*, 2022.
[7] H. Jordan, B. Scholz, P. Subotić, Soufflé: On Synthesis of Program Analyzers, in: Computer Aided Verification, LNCS, Springer, 2016, pp. 422–430.
[8] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, et al., Declarative networking: language, execution and optimization, in: ACM SIGMOD, 2006.
[9] J. M. Hellerstein, P. Alvaro, Keeping CALM: when distributed consistency is easy, CACM 63 (2020) 72–81.
[10] L. Bellomarini, E. Sallinger, G. Gottlob, The Vadalog System: Datalog-based Reasoning for Knowledge Graphs, VLDB 11 (2018) 975–987.
[11] J. Wang, J. Wu, M. Li, J. Gu, A. Das, C. Zaniolo, Formal semantics and high performance in declarative machine learning using Datalog, The VLDB Journal 30 (2021) 859–881.
[12] J. Eisner, N. W. Filardo, Dyna: Extending Datalog for Modern AI, in: Datalog Reloaded, volume 6702 of *LNCS*, Springer, 2011.
[13] T. J. Green, LogiQL: A Declarative Language for Enterprise Applications, in: PODS, 2015.
[14] B. Chin, D. von Dincklage, V. Ercegovac, P. Hawkins, M. Miller, et al., Yedalog: Exploring knowledge at scale, Summit on Advances in Programming Languages (2015).
[15] E. Skvortsov, Logica project, 2023. URL: https://logica.dev/.
[16] E. Skvortsov, Y. Xia, S. Bowers, B. Ludäscher, The Logica System: Elevating SQL databases to declarative data science engines, 2024. URL: https://tinyurl.com/LogicaElevatingSQL.
[17] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, in: Proceedings of the International Conference on World Wide Web, 1998, p. 107–117.
[18] J. Lloyd, R. Topor, Making Prolog more expressive, The Journal of Logic Programming 1 (1984) 225–240.