Logica: Declarative Data Science for Mere Mortals

Evgeny Skvortsov
Google LLC
Kirkland, WA, USA
evgenys@google.com

Yilin Xia
University of Illinois
Urbana-Champaign, IL, USA
yilinx2@illinois.edu

Bertram Ludäscher
University of Illinois
Urbana-Champaign, IL, USA
ludasch@illinois.edu

ABSTRACT
Logica (=Logic + aggregation) is a freely available, open-source, feature-enhanced version of Datalog that automatically compiles logic rules to several SQL platforms, i.e., a lightweight, serverless SQLite engine, a multi-user client-server PostgreSQL system, and a highly scalable, parallel BigQuery instance. Logica combines the beginner-friendly declarative features of Datalog (intuitive, pattern-based queries in the style of QBE), with advanced analytical features needed in data science practitioners when processing large, real-world datasets. The system has been used for data science applications and training in industry, and in graduate-level courses in academia. Logica allows beginners to seamlessly progress from traditional (toy) examples to intermediate and advanced use cases.

1 INTRODUCTION
Datalog has a long and venerable history in databases [16, 17], e.g., as a family of languages for theoreticians studying the expressive power of queries [2], as a logical foundation for non-monotonic reasoning [25, 26], and as a language for teaching the foundations of databases [1]. After much excitement and research activity in the late 1980s and early 1990s, interest temporarily waned1, before it saw a resurgence in academia and industry in the 2000s [7, 12] which continues to this day [3]. Datalog now has many applications, bridging the gap between specification and implementation, e.g., in program analysis [14, 18, 20], declarative networking [13, 15], knowledge graphs [5], and ML/AI [4, 9, 27]. For these and other use cases, a number of specialized Datalog systems and prototypes have been implemented (e.g., see [5, 11, 14] among many others).

Somewhat surprisingly, however, despite these many successful application areas, there has been a lack of freely available, scalable implementations of Datalog that support real-world data science applications. In contrast, Python’s success in data science can be explained by the fact that it is widely available and that it is both, an excellent language for beginners and a production-level language for data science, ML, and AI applications, i.e., 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 would allow beginners to advance from simple, declarative queries to more complex data-intensive analysis use cases.

In this demonstration, we present our answer to this challenge: Logica is a freely available, open-source Datalog variant, designed to combine the declarative features of a logic-based rule language with features needed in real-world data science applications.

1 No practical applications of recursive query theory . . . have been found to date” by Michael Stonebraker in Readings in Database Systems, 3rd Edition is symptomatic.

Figure 1: Logica programs can be executed on a single-user SQLite engine; a multi-user, client-server PostgreSQL system; and the massively parallel BigQuery data warehouse in the cloud.

Logica rules feature both, (i) the traditional positional Datalog syntax, making pattern-based, QBE-style queries intuitive and convenient, especially for low-arity relations (e.g., knowledge graphs represented as triples), and (ii) the name-based syntax known from SQL. This latter feature makes SQL queries, and thus Logica, more robust to schema changes2 and is a practical necessity when dealing with schemas involving more than a handful of columns.

In Section 2 we introduce a few of the key features of Logica by example, starting from simple Datalog queries, to intermediate ones, and ending with more advanced, analytical examples. The use of Logica in practice (e.g., in notebook environments) is discussed in Section 3. In Section 4 we describe the plans for our interactive Logica demonstration: it will feature not only the examples discussed in the next section, but several additional examples that users can execute live, from their own laptops, during the conference. Finally, in Section 5, we summarize the unique features of Logica and our future plans for using Logica in education.

2 LOGICA BY EXAMPLE
2.1 Some Datalog Basics
Datalog rules are implications of the form head :- body, i.e., whenever the conjunction of literals (atoms or their negation) in the body is true, the logic atom in the head is true as well.

2For example, adding new columns to a schema will regularly “break” queries in positional syntax, while queries with name-based syntax often remain unaffected.
Transitive Closure. The following rules derive a new, transitive relation \(\text{Ancestor}(x,y)\) from a given relation \(\text{Parent}(x,y)\). The first rule states that parents are ancestors. The second, recursive rule specifies that a fact \(\text{Parent}(x,z)\) and a derived fact \(\text{Ancestor}(z,y)\) can be joined to derive a new fact \(\text{Ancestor}(x,y)\):

\[
\begin{align*}
\text{Ancestor}(x,y) & : = \text{Parent}(x,y); \\
\text{Ancestor}(x,y) & : = \text{Parent}(x,z), \text{Ancestor}(z,y);
\end{align*}
\]

Note that predicates (relation names) begin with an uppercase letter in Logic, while logical variables are denoted in lowercase. Rules are terminated by a semicolon, as in SQL. Similar to Query-by-Example (QBE) [1, 29], Datalog is a pattern-based language, i.e., multiple occurrences of the same variable in a rule body (e.g., \(z\) in the second rule above) correspond to an implicit join.

Graph Queries. The ease of computing transitive closures indicates that Datalog is a powerful graph query language, sharing features with specialized languages, such as SPARQL and Cypher. Regular path queries (RPQs) [28], can be expressed easily via rules, making Datalog an excellent choice for querying knowledge- and provenance graphs [8]. Assume you want to compute \(\text{LCA}(x,y,a)\), the lowest common ancestor \(a\) of two nodes \(x\) and \(y\) in a given family tree or provenance graph. An elegant, declarative solution is:

\[
\begin{align*}
\text{CA}(x,y,a) & : = \text{Ancestor}(x,a), \text{Ancestor}(y,a), x \neq y; \\
\text{NonLCA}(x,y,a) & : = \text{CA}(x,y,a), \text{CA}(x,y,b), \text{Ancestor}(b,a); \\
\text{LCA}(x,y,a) & : = \text{CA}(x,y,a), \neg \text{NonLCA}(x,y,a);
\end{align*}
\]

The first rule defines a common ancestor \(a\) of two distinct nodes \(x\) and \(y\) (via an implicit join on \(a\)). The second rule finds common ancestors \(a\) for which a "better" (i.e., lower) ancestor \(b\) exists. These non-LCA ancestors are then discarded by the third rule, which uses negation\(^3\) to derive all remaining true LCA's.

2.2 Logica Variables & Named Arguments

The Logica examples above employ the classic positional Datalog syntax, i.e., in which an \(n\)-ary predicate occurrence \(P(x_1, \ldots, x_n)\) must include terms (logical variables or data values) in all argument positions. This facilitates pattern-based queries with implicit joins, and is intuitive and convenient for relations with few attributes (columns) as shown in the graph queries above. However, traditional (non-graph) relations often have many more columns, and the positional syntax becomes inconvenient, error-prone, and unnecessarily brittle (queries regularly break, e.g., when new columns are added to base relations). To this end, Logica also supports named arguments so that queries can only refer to those attributes that are needed to specify the desired query.

Confronting the Real World. In a more realistic setting, the binary \(\text{Parent}(x,y)\) relation above will be populated from a \(\text{Person()}\) table that has many more columns. For simplicity, we use a 4-ary relation with attributes \(\text{name:}, \text{dob:}\) (date-of-birth), \(\text{mother:},\) and \(\text{father:}\) here. Symbols ending in a colon (color-coded green) are attribute names and are often (but not always) followed by data values (e.g., strings) or logic variables.

A base table for \(\text{Person()}\) may include facts of the form:

\[
\begin{align*}
\text{Person(name: } & \text{"Joe Doe", dob: } \text{"1990-08-15"}) \\
\text{mother: } & \text{"Jane Doe", father: } \text{"John Doe"}) \\
\text{Person(name: } & \text{"John Doe", dob: } \text{"1966-05-28"}) \\
\text{mother: } & \text{"Alice Wang", father: } \text{"Robert Doe"}) \\
\end{align*}
\]

From such a base table with named arguments, we can populate the binary parent relation (with positional syntax) as follows:

\[
\begin{align*}
\text{Parent}(x,y) & : = \text{Person(name}:x, \text{mother:mother, father:father}), \\
y & \text{in } \text{[mother,father]};
\end{align*}
\]

The person atom in the body of the rule uses only three attributes (i.e., does not mention \(\text{dob:}\)) and introduces logic variables \(x, \text{mother, and father}\), respectively, representing domain values. The second conjunct in the rule body, i.e., the construct \(y \text{ in [mother,father]}\) effectively specifies a disjunction (the parent \(y\) can be either the mother or the father of \(x\)) and would require two, nearly identical, rules in standard Datalog syntax.

The attribute-variable pattern (e.g., "mother:mother" above) is quite common, so Logica defines a convenient abbreviation: By omitting a logic variable, an implicit variable with the same name as the attribute becomes available. In this way, we can write the previous rule even more concisely as follows:

\[
\begin{align*}
\text{Parent}(x,y) & : = \text{Person(name}:x, \text{mother:,father:}), \\
y & \text{in } \text{[mother,father]};
\end{align*}
\]

Translating Recursive Rules. Several variants and fragments of Datalog exist and they can differ significantly in their handling of recursion and negation, and consequently in their expressive power and computational complexity [1]. Many SQL systems now support a limited form of (linear) recursion through common table expressions (CTEs). When compiling Datalog to SQL there are two natural approaches: (1) employ the WITH RECURSIVE construct and CTEs, or (2) unroll recursive rules to a fixed depth \(k\). Both approaches have advantages and disadvantages, e.g., (1) cannot handle non-linear recursion, while (2) may result in incomplete answers when the recursion depth required exceeds the fixed "unroll-depth" \(k\) set for a given program. Experience with Logica’s approach (2) has rarely presented a real limitation: A double-recursive variant of the \(\text{Ancestor}\) program above, e.g., can handle chains of length \(O(2^k)\) given an unroll-depth of \(k\).

2.3 A Data Science Use Case

Consider a data engineer working for the Seattle Public Library (SPL), who is tasked with analyzing loan data for decision support and new library services. She accesses the external \(\text{LibraryLoans()}\) table via the library’s PostgreSQL database, and begins to explore the data in her Jupyter notebook. First, she wants to visualize the data in her Jupyter notebook. First, she wants to visualize the types of materials (books, e-books, DVDs, etc.) that are popular, i.e., having more than 30K checkouts per type. For materials which have no checkouts in a given month, missing values should be padded with zeros, to keep the plotting code simple.

\[
\begin{align*}
\text{LoansByMonth(material: PopularMat(), month: AnyMonth())} & \equiv 0; \\
\text{LoansByMonth(material:, month:) := loans :- LibraryLoans(material:, month:,loans:), material = PopularMat();} \\
\text{PopularMat() := material: MatCount(material) > 30000;} \\
\text{MatCount(material) := 1 := LibraryLoans(material:);} \\
\text{AnyMonth()} & \equiv 1 + 1 := \text{in Range(12)};
\end{align*}
\]

She defines a new result relation \(\text{LoansByMonth()}\) that aggregates loans by material-type and month, for popular materials. The function \(\text{PopularMat()}\), defined in \(L6\) and used in \(L1\) and \(L4\) includes only materials with more than 30K checkouts total. \(\text{MatCount()}\) sums up checkouts for each material-type. The external \(\text{LibraryLoans()}\) contains counts of loans, grouped by material

\[
\begin{align*}
\text{LoansByMonth(material: PopularMat(), month: AnyMonth())} & \equiv 0; \\
\text{LoansByMonth(material:, month:) := loans :- LibraryLoans(material:, month:,loans:), material = PopularMat();} \\
\text{PopularMat() := material: MatCount(material) > 30000;} \\
\text{MatCount(material) := 1 := LibraryLoans(material:);} \\
\text{AnyMonth()} & \equiv 1 + 1 := \text{in Range(12)};
\end{align*}
\]

She defines a new result relation \(\text{LoansByMonth()}\) that aggregates loans by material-type and month, for popular materials. The function \(\text{PopularMat()}\), defined in \(L6\) and used in \(L1\) and \(L4\) includes only materials with more than 30K checkouts total. \(\text{MatCount()}\) sums up checkouts for each material-type. The external \(\text{LibraryLoans()}\) contains counts of loans, grouped by material
and month. The multi-valued function AnyMonth() returns integers 1, . . . , 12 matching the representation used in LibraryLoans() for the months of the year.

Figure 2 depicts the dataflow for the result relation (with diamond shape) LoansByMonth() of the SPL example above. This graph is automatically generated by Logica, when the program is run in a Jupyter notebook. The output depends on the external database table LibraryLoans() in PostgreSQL and intermediate grounded functional predicates (rounded boxes) MatCount() and PopularMat().

Migrating from PostgreSQL to BigQuery. In our demonstration scenario [23], the engineer is happy with the result, but notices that query evaluation took more than 15 seconds. While this is fine for an initial exploration, a deeper analysis on an even larger volume has been requested, so the engineer changes the Logica directive @Engine("psql") in the notebook to @Engine("bigquery"). After pushing the data to Google’s scalable, distributed BigQuery data warehouse, she is pleased to see that once the data is loaded to BigQuery the very same Logica rules now take less than 2 seconds to execute.4

4 DEMONSTRATION PLAN

Core Demo. After a brief introduction and motivation, our Logica demonstration will include the following steps:

(1) A gentle introduction to declarative (Datalog) querying with Logica, using classic examples such as Ancestor(x, y) (for computing transitive closures), intermediate graph queries such as LCA(x, y), and their application for querying provenance [8] and knowledge graphs via RPQs [28].

(2) Intermediate examples that highlight the commonalities and differences between standard Datalog rules and systems and Logica: programmer-friendly syntax enhancements (for mixing positional and named arguments), data science extensions for user-defined functions and aggregation (e.g., the SPL example above), composite data types, and notebook integration.

(3) An advanced data science use case (expanding on the SPL example) that involves multiple SQL engines simultaneously to illustrate data import/export and performance differences.

Go Logica and Logica-to-Go. In addition to giving a live demo of the Logica system using interactive notebooks ("Go Logica"), we plan to invite conference attendees to participate in the demonstration session by running Logica notebooks in the cloud through Google colab [23]. In this way, participants will get a first-hand experience of the power of declarative rules for data science.

During the conference we will also provide a "Logica-to-Go" clinic for interested participants, i.e., a 5–10 minute installation walk-through whose goal is to empower participants to run their own instance of Logica with Python and SQLite. We believe that many attendees will have Python 3—and thus also SQLite—pre-installed on their laptop computers, making installation of a minimal, SQLite-based Logica system straightforward. By leaving

4The SPL dataset is about 3 GB in size and has 14 million rows [23].
the demo-session with a self-contained Logica installation and beginner’s tutorial, participants will be independent of online and cloud resources, and will be able to experiment on their own laptops with declarative querying and data analysis examples, whether on their way home from the conference with unreliable internet access, or cast away on an island like Robinson Crusoe.

5 CONCLUSIONS

Datalog has had a long and venerable (some might say turbulent) history in the database community [16, 17]. Database theory [1] cannot be imagined without it, but it is the many new (or rediscovered) applications that have led to a resurgence of Datalog in academia and industry in the last 10–20 years. Despite the more abundant and novel use cases, and the continued interest in declarative, Datalog-like languages in academia and industry, it is somewhat surprising that there is a shortage of freely available, scalable Datalog systems, i.e., which can handle practical database and data science applications, in addition to the usual toy examples encountered in a database theory course.

Powerful Datalog engines have been developed in industry and research labs (e.g., see [5, 6, 11] among many others [16]), but these are often not freely available, can be hard to install and use, or may even have been abandoned. Instructors who are eager to teach foundations of databases, declarative programming, or new applications [12, 13] may also find it difficult to identify suitable systems. While there are some excellent open-source systems (e.g., for teaching relational languages [24]), and specialized systems (e.g., for answer set programming [10]), these are generally not suitable for data-intensive or data science applications.

The first author has developed Logica, a powerful open-source tool that combines the elegance and declarative expressiveness of logic rules, with down-to-earth practical language features, e.g., for user-defined functions, aggregation, and complex data types that are necessary for real-world data science applications [22]. The core of the system is a source-to-source compiler that generates SQL code for different systems, currently SQLite, PostgreSQL, and BigQuery. With our demonstration, and using various examples, we will illustrate that there is a common path, from beginner to expert analyst, that data science students and practitioners can follow, all while staying in the framework of declarative, rule-based data science.

The last author has extensive experience teaching database and information science courses at the undergraduate and graduate level, and has recently begun, assisted by the second author, to use Logica for teaching, e.g., as a declarative modeling and query language, and for checking and enforcing integrity constraints (in the context of data cleaning and repair).

Plans for the Future. We believe that Logica fills an important practical need by combining the powerful features of a declarative rule language with the robust features of well-engineered SQL database systems, resulting in a novel hybrid language that supports a wide range of users and uses. Logica’s ability to handle large, real-world datasets provides a promising avenue for teaching, and we plan to develop new course materials using Logica in the future. We are also planning Logica extensions (or a preprocessor) to automatically translate RPDQs to Datalog—e.g., similar to [8]—to better support use cases involving provenance and knowledge graphs. For OLAP-style workloads we are also considering to add DuckDB [19] as a new target platform.

REFERENCES