Between Two Worlds: Logica Intertwines ASP and SQL

Evgeny Skvortsov[1], Yilin Xia[2], Ojaswa Garg[1], Shawn Bowers[3] and Bertram Ludäscher[2]

[1]Google LLC, USA
[2]University of Illinois, Urbana-Champaign, USA
[3]Gonzaga University, USA

Abstract

Answer Set Programming excels at combinatorial search: generating models, solving constraints, and optimizing over discrete choices. SQL-based evaluation excels at data analysis: aggregation, joining, and transforming large datasets. Many real-world problems require both a search phase that explores a combinatorial space, followed by an analytical phase that examines the results, or a pipeline where data preparation feeds into constraint solving and the solutions are analyzed further. We present an integration of ASP into Logica, an open-source logic programming language that compiles to SQL. We introduce `couldbe`, `cantbe`, and `shouldbe` operators that extend Logica's expressive power with choice rules, integrity constraints, and optimization objectives, respectively. The programmer writes a single Logica program and the system compiles the deterministic fragment to SQL (executed by DuckDB) and the non-deterministic fragment to an ASP program (solved by Clingo). Solutions returned by Clingo are materialized as SQL tables, available within the same program for further analysis using Logica's full analytical power: aggregation, recursive evaluation, and functional composition. The program may contain an arbitrary number of non-deterministic fragments. We illustrate the integration through three examples that combine search and analysis in ways that neither paradigm handles well alone: identifying strategically essential companies in a supply chain, analyzing adversarial robustness of graph coloring, and detecting cliques in a social network and measuring their influence via PageRank. The integration is implemented and publicly available as part of the Logica system.

Keywords

Answer Set Programming, Logic Programming, SQL, Logica, Clingo, Multi-Paradigm

CEUR-WS.org/Vol-4311/ASPOCP_regular_10.pdf

1. Introduction

Answer Set Programming and SQL represent two of the most successful declarative paradigms in computing. ASP, built on stable model semantics [14], excels at combinatorial search: given a space of possibilities, it finds models that satisfy constraints and optimize objectives. SQL, backed by decades of database engineering, excels at data analysis: aggregation, joining, and transforming datasets at scale. Each paradigm has mature implementations, with Clingo [13] for ASP and engines like DuckDB [20] and PostgreSQL for SQL, and each is widely applied to problems within its respective domain.

Yet many problems live between the two worlds. A social-network analyst may need to detect hidden cliques, a combinatorial search problem, and then measure how their removal reshapes influence rankings, a data analytics problem. A supply-chain auditor may need to enumerate all valid control sets and then determine which companies are truly essential. A network security engineer may need to test whether a network's resource allocation, modeled as a graph-coloring problem, survives every possible adversarial perturbation. In each case, combining both paradigms today means stitching together separate tools: write an ASP program, export the solutions, import them into a database, write SQL queries, and manually manage the pipeline.

Logica [26, 25] is an open-source logic programming language that compiles to SQL, offering a natural bridge between the two worlds. A Logica program is a collection of rules written in Datalog-style syntax, extended with aggregation (including through recursion) and functional notation. The Logica compiler translates program rules into SQL queries executed by a backend engine, with support for DuckDB, PostgreSQL, SQLite, and BigQuery. Because Logica combines Datalog-style syntax with SQL

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal.



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

compilation, it can host both paradigms under one roof: the deterministic fragment stays in SQL, while the non-deterministic fragment is delegated to a solver.

In this paper, we extend Logica with three new operators for ASP integration, `couldbe`, `cantbe`, and `shouldbe`, that qualify Logica rules for non-deterministic evaluation. These operators use standard Logica rule syntax but signal to the compiler that the associated rules should be compiled to ASP rather than SQL:

- `couldbe` designates a rule as a choice, where the solver may or may not include each derived fact in the answer set;
- `cantbe` designates a rule as an integrity constraint, where any model in which the body holds is rejected; and
- `shouldbe` designates a rule as an optimization objective, where the solver seeks answer sets that maximize or minimize the given expression.

The programmer writes a single Logica program mixing deterministic and non-deterministic rules. Each invocation of the solver is triggered by a `Clingo` call that lists the predicates to compile to ASP; the system checks that all dependencies of the listed predicates are included. The designated rules are compiled to an ASP program solved by Clingo, while the remaining rules compile to SQL executed by the underlying database engine (where current support for the ASP extensions is limited to DuckDB). Solutions returned by Clingo are materialized as SQL tables, immediately available for further analysis using aggregation, recursion, and functional composition.

We illustrate the integration through three examples, each combining ASP-based search with SQL-based analysis in Logica:

- **Strategic companies.** Clingo enumerates all valid control sets in a supply chain and Logica filters to minimal sets and ranks each company by strategic importance.
- **Adversarial coloring.** Logica generates all possible edge-addition attacks on a graph, and for each attack, Clingo tests whether a 3-coloring still exists. Logica then analyzes which attacks succeed in breaking the coloring.
- **Conspiracy detection.** PageRank identifies influential nodes in a social network and Clingo finds a maximum packing of disjoint cliques. Logica then collapses each clique into a single node and recomputes the rankings to reveal hidden influence.

Each example requires both combinatorial search and data analysis, tightly integrated within a single program. The integration is implemented and publicly available as part of the Logica system.[1]

The rest of the paper is organized as follows. Section 2 gives a brief overview of Logica. Section 3 describes the ASP integration and the three operators. Sections 4–6 present the three examples. Section 7 discusses related work, and Section 8 concludes.

2. Logica

Logica is an open-source logic programming language that compiles to SQL. Programs are collections of rules in Datalog-style syntax. Predicates begin with uppercase letters, variables begin with lowercase letters, and statements (facts and rules) are terminated by semicolons. The compiler translates rules into SQL queries for execution by a backend engine.

Logica extends classical Datalog in several ways relevant to this paper. *Named arguments* allow predicates to use field names instead of positional arguments:

```
Edge(from: "a", to: "b", weight: 3);
```

A call only needs to mention the fields it uses, and the rest are projected away: `Edge(from: x)` matches every edge out of `x`, regardless of its `to` and `weight` values.

Functional notation lets predicates return values:

[1] https://logica.dev

```
Square(x) = x * x;
Hypotenuse(a, b) = Sqrt(Square(a) + Square(b));
```

Aggregation operators accumulate values across matching rows. The operator += computes a sum, `Min=` a minimum, `List=` a list, and so on:

```
Degree(x) += 1 :- Edge(from: x);
ShortestPath(x) Min= d :- Path(x, d);
```

The same operators also work inside expressions, written `Op{value :- condition}`, e.g., `Count{b :- C(a, b)}` counts the colors assigned to `a`.

Several additional features are used in the examples throughout this paper. The `distinct` keyword specifies that a predicate produces a set (no duplicates), corresponding to SQL's `GROUP BY` over all columns. For example, `Node(x) distinct` collects unique nodes. Negation is written `~P(x)` and holds when `P(x)` is not derivable (negation as failure). The `in` operator supports list membership testing and iteration, as in `x in [a, b]`, which binds `x` to each element of the list. Logica also supports records, written as `{field: value, ...}`, for representing structured data with named fields. Record fields are accessed using dot notation (e.g., `r.field`).

The following PageRank example illustrates how these features work together. PageRank is a recursive computation that divides a node's rank in equal shares among its neighbors and itself:

```
Node(x) distinct :- G(a, b), x in [a, b];
D(x) += 1 :- G(x, y);
NumNodes() += 1 :- Node();
ResetP() = 0.1;
WalkP() = 1 - ResetP();

PR(x) += ResetP() / NumNodes() :- Node(x);
PR(y) += PR(x) * WalkP() / (D(x) + 1) :- G(x, y);
PR(x) += PR(x) * WalkP() / (D(x) + 1);
```

Each += rule contributes additively to the PageRank of a node. The first += rule seeds every node with uniform probability; the second distributes rank along edges; the third retains a share for the node itself. The second and third rules both carry the damping factor `WalkP()`, so each node passes on exactly `WalkP()` of its rank and total mass is conserved. Logica compiles these rules into SQL aggregation queries, executed iteratively by the Pipeline Driver until convergence (see Section 3.2). This PageRank definition will reappear in Section 6, where we reuse it via Logica's *functor* mechanism.

Functors allow a predicate's dependencies to be parameterized. If `PR` is defined in terms of a graph `G`, we can instantiate it on a different graph with a single line:

```
PRC := PR(G: CleansedG);
```

This creates a new predicate `PRC` that computes PageRank over `CleansedG` instead of `G`, reusing the same definition without duplication.

3. ASP Integration

This section describes the Logica-ASP integration. We first describe the three new operators that designate Logica rules for non-deterministic evaluation (Section 3.1). We then describe the system architecture and implementation for compiling and executing these rules within Logica (Section 3.2), ending with a summary of the overall compilation process (Figure 2).

3.1. The Three Operators

The operators read as English and correspond directly to ASP constructs. As a running example, consider a knapsack problem that selects items to maximize total value without exceeding a weight limit. The following Logica program solves this using the `couldbe`, `cantbe`, and `shouldbe` operators.

```
# Item facts: name, weight, value.
Item("Water", 3, 20);
Item("Food", 2, 15);
Item("Rope", 1, 5);

# Choice: each item may or may not be selected.
Selected(item) couldbe :- Item(item, weight, value);

# Constraint: total weight must not exceed 5.
Overweight() cantbe :- ISum{item -> weight :- Selected(item), Item(item, weight, value)} > 5;

# Compute total value of selected items.
TotalValue(total) :- total = ISum{item -> value :- Selected(item), Item(item, weight, value)};

# Objective: maximize total value.
Optimal() = Max{v :- TotalValue(v)} shouldbe;

# Invoke Clingo with all predicates.
Models() = Clingo(["Item", "Selected", "Overweight", "TotalValue", "Optimal"], []);
```

The choice operator `couldbe` means the fact may or may not hold: each item could be selected, or not. The compiler translates this into an ASP choice rule [24], which places no cardinality bound on the selection. The constraint operator `cantbe` means being overweight is forbidden: any answer set where the body holds is rejected. The compiler translates this into a headless integrity constraint. The optimization operator `shouldbe` means the value should be as large as possible. Here Logica plays a deliberate trick with its syntax: the aggregation operators `Max` and `Min`, normally used for computing maxima and minima over data, become optimization directives when combined with `shouldbe`. Writing `Max{...} shouldbe` compiles to `#maximize`; `Min{...} shouldbe` compiles to `#minimize`. The same keyword serves different purposes depending on context: aggregation in deterministic rules compiled to SQL, optimization in non-deterministic rules compiled to ASP.

The remaining predicates (`Item` and `TotalValue`) are ordinary deterministic rules. They compile to plain ASP facts and rules. The programmer uses the same Logica syntax throughout; only the three operators signal which rules carry non-deterministic intent. Rules compiled to ASP can also use negation as failure (`~P(x)` becomes `not p(X)`), disjunction (`|` produces multiple ASP rules), and aggregation (`Sum`, `Count` become `#sum`, `#count`). The `ISum` (individualized sum) variant used in the knapsack listing includes discriminating keys in the aggregate tuple (`#sum { V,K : ... }`), preventing collapse of distinct items that share the same value. These features appear throughout the examples of Sections 4–6.

The program above compiles to the following ASP:

```
% From: Item("Water", 3, 20); ...
item("Water",3,20).
item("Food",2,15).
item("Rope",1,5).

% From: Selected(item) couldbe :- ...
{ selected(ITEM) } :- item(ITEM,WEIGHT,VALUE).

% From: Overweight() cantbe :- ...
:- #sum { WEIGHT,ITEM : selected(ITEM), item(ITEM,WEIGHT,VALUE) } > 5.

% From: TotalValue(total) :- ...
total_value(TOTAL) :-  TOTAL = #sum { VALUE,ITEM : selected(ITEM), item(ITEM,WEIGHT,VALUE) }.

% From: Optimal() = Max{...} shouldbe;
#maximize { V : total_value(V) }.
```

Each Logica rule maps to one ASP rule (or, where body disjunction is used, to one rule per disjunct), with the operator determining the form. Logica's PascalCase predicate names become snake_case in the generated ASP.

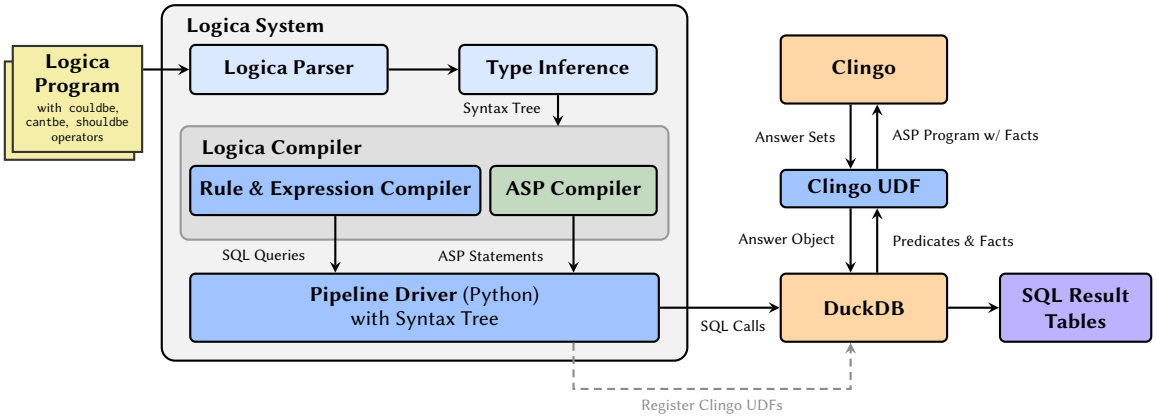


Figure 1: Architecture of the Logica/Clingo integration, extending the Logica system architecture [26]. The Logica compiler produces both SQL queries (via the Rule & Expression Compiler) and ASP statements (via a new ASP Compiler). The Pipeline Driver executes SQL on DuckDB and registers the Clingo UDFs, which compile predicates and facts into a complete ASP program, invoke the Clingo solver, and return answer sets as nested DuckDB values for further SQL-based analysis.

3.2. Architecture and Implementation

This section describes how the Logica system compiles and executes programs that mix deterministic and non-deterministic rules. We first review the standard Logica pipeline, then describe the extensions for ASP support. Figure 1 shows the overall architecture, and Figure 2 summarizes the compilation and execution process.

Logica System Architecture. The standard Logica compilation process [26] consists of three main stages. The *parser* reads the source program (and any imported modules) and builds a syntax tree. The *type inference engine* analyzes this tree, assigning types for error detection and dialect-specific SQL generation. The *compiler* (consisting of a Rule Compiler and an Expression Compiler) translates the type-annotated program into SQL queries. For programs without recursion, the compiler produces a single self-contained SQL script. For recursive programs, the programmer can specify a fixed iteration count, which the compiler unrolls into a single SQL script, or use iterative evaluation, where the *Pipeline Driver* repeatedly executes the recursive queries until a termination condition is met. Each iteration replaces the prior state entirely. This replacement semantics is key to Logica's support for non-monotonic computation, where derived values may change between iterations (e.g., shortest path distances that decrease as shorter paths are found).

Extensions for ASP Support. The Logica-ASP integration adds a new *ASP Compiler* that translates rules annotated with `couldbe`, `cantbe`, and `shouldbe` operators into ASP statements. These statements, together with the program's syntax tree, are captured by the Pipeline Driver. At initialization, the Pipeline Driver registers several Clingo-related user-defined functions (UDFs) with DuckDB, including `Clingo()` for invoking the solver, `ExtractClingoCall()` for unpacking answer sets, and `CompileClingo()` for debugging the generated ASP program. When DuckDB encounters a `Clingo()` call during SQL evaluation, the UDF compiles the requested predicates into a complete ASP program (combining the ASP statements with ground facts assembled from the SQL side), invokes the Clingo solver, and returns the answer sets as a nested DuckDB value. The integration is enabled with an engine directive, specified directly in the Logica program. The following example directive selects DuckDB as the backend SQL engine and configures Clingo with no time limit and no limit on the number of answer sets returned:

```
@Engine("duckdb", clingo: {time_limit: inf, models_limit: inf});
```

Working with ASP Fragments. The programmer explicitly selects which predicates constitute the ASP fragment by listing them in the `Clingo()` call. This design is deliberate: rather than automatically partitioning the program, it gives the programmer control (and responsibility) over exactly which rules are compiled to ASP and which are compiled to SQL. The system validates that all dependencies of the selected predicates are included, producing an error if a required predicate is missing. This prevents subtle bugs where a predicate is accidentally evaluated in SQL when it was intended for ASP, or vice versa. Multiple `Clingo()` fragments can be layered, one consuming another's answer sets. Mutual recursion across the boundary is evaluated by Logica's iterative engine following its recursion semantics [27], and thus might not agree with the ASP semantics of combining the fragments into a single ASP program; the one-shot compilation path does not yet support such cycles.

The Clingo Function. The `Clingo` function is implemented as a Python closure that captures the syntax tree of the entire Logica program. From DuckDB's perspective it is an ordinary scalar function that takes arguments and returns a value. Internally, it carries the full program and can compile any fragment of it into ASP on demand. The function takes two arguments: a list of predicate names that identify which rules should be compiled to ASP, and a list of ground facts that provide data from the SQL side. The facts can be an empty list (as in the knapsack example above), a list of explicitly constructed `ClingoFact` values, or generated dynamically via a list comprehension. In this example, drawn from the conspiracy detection program in Section 6, a list comprehension converts each edge in the graph `G` into an ASP fact, providing the input data for the clique-finding ASP fragment:

```
Models() = Clingo(["Clique", "N", "TooSmall", "NotClique", "E", "Paranoia", "ConspiratorsCount"],
                  List{ClingoFact("E", [a, b]) :- G(a, b)});
```

When DuckDB evaluates the SQL query containing this call, the closure looks up the named predicates in its copy of the syntax tree and compiles them into an ASP program. During compilation, `couldbe` rules become choice rules, `cantbe` rules become integrity constraints, and `shouldbe` rules become optimization statements. The closure then appends the ground facts and invokes Clingo.

The predicate names select *which* fragment of the Logica program becomes the ASP program. The facts provide *data* from the SQL side such as a graph to color, a set of edges, or any other structure that Logica has already computed. This separation is the heart of the integration: the non-deterministic logic lives in the program's syntax tree, while the data it operates on is assembled at query time.

Results and Extraction. Clingo returns answer sets, and the function packages them into a single DuckDB value consisting of a list of models, where each model is a list of ground atoms identified by a `model_id`. This value flows back into the SQL query like any other function result, representing a set of possible worlds where each world is a set of ground atoms; atoms not in the set are false. Concretely, the return type is a DuckDB nested structure consisting of a list of records, where each record contains a `model_id` (an integer identifying the answer set) and a `model` (a list of ground atoms, each represented as a predicate name and a list of string arguments). This structured representation allows DuckDB's standard list and record operations to be used for filtering, joining, and aggregating across answer sets. The `model_id` ordering reflects Clingo's enumeration order, with the last model being optimal when optimization is used.

Since `Clingo` is a scalar function, it can be parameterized. In the adversarial coloring example (Section 5), writing `Models(attack) = Clingo(. . . )` causes DuckDB to call the function once per attack value, coordinating hundreds of independent solver invocations automatically.

Results are unpacked with `ExtractClingoCall`, which selects atoms from the answer object by predicate name and model identifier. In this example, the extraction is bound to the answer object stored in `Models()` and retrieves all `Selected` atoms from each answer set, creating a `Solution` predicate indexed by `model_id`:

```
Solution(item, model_id:) :- ExtractClingoCall(item, predicate: "Selected", model_id:) = Models();
```

Extracted facts become ordinary Logica predicates, materialized as SQL tables and available for aggregation, joins, and further analysis.

Algorithm: Logica/Clingo Compilation and Execution

Input: Logica program P with couldbe/cantbe/shouldbe rules
Output: SQL result tables

$T \coloneqq \mathsf{Parse}(P)$ // generates a program syntax tree
$T^\star \coloneqq \mathsf{TypeCheck}(T)$ // type annotations added to T
$Q \coloneqq \mathsf{CompileSQL}(T^\star)$ // deterministic rules → SQL queries
$A \coloneqq \mathsf{CompileASP}(T^\star)$ // non-deterministic rules → ASP statements
$C \coloneqq \mathsf{CreateDuckDBConnection}()$ // in-process SQL backend
$\mathsf{RegisterClingoUDFs}(C, T^\star, A)$ // solver becomes a scalar UDF
foreach *query* Q_i *in* Q **do** // in dependency order
  Execute Q_i on C // evaluated by DuckDB
  if Q_i *contains a call* `Clingo`$(preds, facts)$ **then** // one call per problem instance
    $P_{\mathsf{asp}} \coloneqq A[preds] \cup \mathsf{Render}(facts)$ // selected rules + ground facts
    $S \coloneqq \mathsf{SolveClingo}(P_{\mathsf{asp}})$ // enumerates answer sets
    Package S as nested DuckDB value // builds a record per answer set
    Return it to Q_i // answer sets become queryable data
  end
end
Materialize results as SQL tables // one table per predicate

Figure 2: Overview of the compilation and execution pipeline. The program is parsed, type-checked, and compiled into SQL queries (deterministic rules) and ASP statements (non-deterministic rules). The Pipeline Driver registers the Clingo UDFs with DuckDB and executes the SQL queries. When a query invokes the `Clingo` function, the UDF assembles a complete ASP program from the compiled statements and ground facts, invokes the Clingo solver, and passes the answer sets back as a nested DuckDB value for further SQL-based analysis.

Debugging. For convenience of debugging, Logica provides the `CompileClingo()` function, which returns the generated ASP program as text (as shown above for the knapsack example). Since the boundary between the two engines is explicit, the programmer can separately inspect the problem formulation (via `CompileClingo`), the solver's output (via the answer-set predicates), and the downstream analysis (as standard Logica/SQL).

4. Strategic Companies

The strategic companies problem [19] is a classic ASP benchmark. Companies produce goods and are controlled by other companies. A set of companies is a *control set* if (1) every product has at least one producer in the set, and (2) every company whose controllers are all in the set is also in the set. In the classical formulation [8, 19], a company is *strategic* if it belongs to *some* minimal control set; deciding this *brave* membership question is Σ_2^P-complete [8]. Our example computes the natural *cautious* variant (the companies that appear in *every* minimal control set), whose membership problem is the Π_2^P-complete dual. Both readings sit at the second level of the polynomial hierarchy, and in our integration they are one SQL predicate apart: because the whole set of minimal control sets is materialized as a table, the cautious test `num_models == NumMinimalModels()` becomes the brave one by writing `num_models >= 1`. This makes the problem a natural benchmark for disjunctive ASP, where answer-set minimality captures subset-minimality directly. Logica has no head disjunction, so we take the complementary route: a choice rule generates all control sets and minimality is recovered on the SQL side. This enumerates non-minimal sets as well, but yields all minimal models as an ordinary relation, which further analyses (e.g., counting or ranking) can query directly. In our integration, this two-level structure becomes architecturally explicit: Clingo explores the space of control sets, and Logica handles the quantification over solutions, filtering to minimal sets and verifying membership across them.

This example follows an enumeration pattern: Clingo generates all answer sets, and Logica analyzes the collection. Figure 3 shows the complete program. The instance data uses the standard encoding

from [19]: each product is produced by at most two companies (`ProdBy`) and each company is controlled by at most three others (`ContrBy`). These fixed arities are a presentation simplification; the ASP constraints enforce the corresponding readings: no product may lack a producer in the control set, and every company whose controllers are all in the set must itself be included.

```
# Production and control relationships.
ProdBy("meat", "Northern Butcheries", "Southern Grind");
ProdBy("cabbage", "Peaceful Joe", "Peaceful Joe");
ProdBy("spacetravel", "Stargazer", "Stargazer");
ProdBy("toys", "Toys Unlimited", "Gonzaga Toys");
ProdBy("shovels", "Open Kopai", "DeltaShovel");

ContrBy("Northern Butcheries", "Southern Grind", "Peaceful Joe", "Stargazer");
ContrBy("Toys Unlimited", "Stargazer", "Stargazer", "Stargazer");
ContrBy("Gonzaga Toys", "Open Kopai", "DeltaShovel", "Stargazer");

Company(c) :- ProdBy(x, y, z), c in [y, z];

##### ASP: select a control set #####
Control(c) couldbe :- Company(c);
LackingProduct() cantbe :- ProdBy(p, x, y), ~Control(x), ~Control(y);
LackingControl() cantbe :-
  ContrBy(c, x, y, z), Control(x), Control(y), Control(z), ~Control(c);

# Enumerate all control sets.
Models() = Clingo(
  ["ProdBy", "ContrBy", "Company", "Control",
   "LackingProduct", "LackingControl"], []);

# Extract results: which companies are in each model.
ControlInModel(company, model_id:) :-
  ExtractClingoCall(company, predicate: "Control", model_id:) = Models();
CompaniesInModel(model_id) List= company :- ControlInModel(company, model_id:);

# Find minimal control sets.
Subset(c1, c2) :- Size(Set{x :- x in c1 ++ c2}) == Size(c2);
DefeatedModel(model_id) :-
  c1 = CompaniesInModel(model_id), c2 = CompaniesInModel(other_model),
  model_id != other_model, Subset(c2, c1);
MinimalModel(model_id) distinct :- ControlInModel(model_id:), ~DefeatedModel(model_id);
NumMinimalModels() += 1 :- MinimalModel();

# Count strategic importance.
@OrderBy(StrategicInModel, "model_id", "col0");
StrategicInModel(company, model_id:) :-
  ControlInModel(company, model_id:),
  MinimalModel(model_id);
StrategicCount(company:, num_models? += 1) distinct :-
  StrategicInModel(company, model_id:);

# Identifying strategic companies.
Strategic(company) :-
  StrategicCount(company:, num_models:),
  num_models == NumMinimalModels();
```

Figure 3: Strategic companies: enumerating and analyzing control sets.

The ASP fragment is concise: `couldbe` makes each company a candidate for the control set, and two `cantbe` rules enforce the closure conditions. With `models_limit` set to allow all solutions, Clingo enumerates every control set.

The analytical work happens in Logica. Each answer set is a list of companies. `Subset(c1, c2)`

checks whether `c1` is a subset of `c2` by concatenating the two lists (`c1 ++ c2`), collecting the result into a set using a set comprehension (`Set{...}`), and comparing the size to `c2`. If the size is unchanged, all elements of `c1` were already in `c2`. `DefeatedModel` identifies any control set that properly contains another. What remains are the minimal control sets. Finally, `StrategicCount` counts how many minimal models each company appears in, using the `?` suffix to indicate `num_models` as an aggregated column, and `Strategic` then selects the companies whose count equals `NumMinimalModels()`, those appearing in every minimal control set.

This example highlights a strength of the integration: ASP generates the solution space, but the comparative analysis across solutions (set containment, minimality filtering, counting) is expressed naturally in Logica's aggregation-rich SQL.

5. Adversarial Graph Coloring

Given a 3-colorable graph and a set of candidate edges that an adversary may add, how robust is the coloring? The adversary selects a pair of edges to insert; if the augmented graph has no valid 3-coloring, the adversary wins. We want to determine which attacks succeed. The problem has a $\exists\forall$ structure: the adversary existentially picks an attack, and for the attack to succeed every possible coloring must fail. An attack is a pair of candidate edges, so there are only $\binom{|\texttt{GE}|}{2}$ of them, and the program answers the question with one coNP check (one Clingo call) per attack; the problem therefore lies in P^{NP}, not at the second level of the polynomial hierarchy. The division of labor is natural: Logica enumerates the attacks and aggregates the results, and Clingo checks each attack for a surviving coloring.

```
# Given a graph G(x, y) and adversarial edge candidates GE(x, y).

# Enumerate all attacks (pairs of adversarial edges).
Attack() = [{a: a1, b: b1}, {a: a2, b: b2}] :-
  GE(a1, b1), GE(a2, b2),
  [a1, b1] <= [a2, b2],
  a1 != a2 || b1 != b2;

Color("red"); Color("green"); Color("blue");

##### ASP: 3-coloring #####
C(a, b) couldbe :- Node(a), Color(b);
Improper() cantbe :-
  C(a1, c), C(a2, c), Edge(a1, a2)
  | Node(a), Count{b :- C(a, b)} != 1;

# Construct augmented graph for each attack.
InstanceG(x, y, attack: Attack()) :- G(x, y);
InstanceG(x, y, attack:) :- {a: x, b: y} in attack, attack = Attack();
InstanceN(x, attack:) distinct :- x in [a, b], InstanceG(a, b, attack:);

# Note: predicates Node and Edge are passed as facts via the second argument.
# Parameterized Clingo: one call per attack.
Models(attack) = Clingo(
  ["C", "Color", "Edge", "Node", "Improper"],
  List{ClingoFact("Edge", [a, b]) :- InstanceG(a, b, attack:)}
  ++ List{ClingoFact("Node", [a]) :- InstanceN(a, attack:)}) :-
  attack = Attack();

# Analysis: which attacks break the coloring?
OpponentWin(attack) :- Size(Models(attack)) == 0;
ModelsStats(num_models:, num_attacks? += 1) distinct :-
  Models() = models, num_models = Size(models);
```

Figure 4: Adversarial graph coloring: testing robustness under edge additions.

The program of Figure 4 has a clear division of labor. Logica handles everything deterministic: generating the graph, producing adversarial edge candidates, and enumerating all $\binom{|\texttt{GE}|}{2}$ possible attacks. For each attack, it constructs the augmented graph (`InstanceG`) and passes it to Clingo via `ClingoFact`. `Attack()` is a functional predicate with a rule body, so it yields one list per attack rather than a single constant, and `attack = Attack()` binds `attack` to each in turn.

The ASP fragment is generic: it simply defines what a proper 3-coloring is. The attack-specific graph arrives as `Edge` and `Node` facts.

The key line is `Models(attack)`: Clingo is invoked once per attack value, and the system coordinates potentially hundreds of solver calls automatically. After solving, Logica analyzes the aggregate results: `OpponentWin` identifies successful attacks, and `ModelsStats` produces a distribution over how many colorings survive each attack. Positional arguments are shorthand for the field names `col0`, `col1`, and so on, so they can be omitted like any other field. The call `Models()` therefore projects away the `attack` argument, giving `ModelsStats` one row per attack.

6. Conspiracy Detection

Consider a social network where some nodes form a tightly connected clique, a "conspiracy." Individually, the conspirators may not rank highly, but collectively they amplify each other's influence. We want to find disjoint cliques (conspiracies) covering as many nodes as possible, neutralize them by collapsing each into a single representative, and measure how the influence rankings change. This example demonstrates how the integration combines the strengths of both paradigms: Clingo solves an NP-hard problem (maximum clique packing), while Logica handles efficient iterative computation (PageRank) that would be unnatural to express in ASP.

This example follows a pipeline pattern: Logica computes PageRank, Clingo finds the maximum clique packing, and Logica analyzes the result.

The program of Figure 5 has three phases. First, Logica builds the graph and computes PageRank using recursive aggregation (`+=`). Second, the ASP fragment searches for disjoint cliques that together maximize the total number of conspirators: the `couldbe` rule generates candidate clique memberships, `cantbe` rules reject cliques with fewer than four members or violate the clique property, and `shouldbe` maximizes the total number of conspirators. Each clique is identified by a *leader*; the last `NotClique` disjunct requires each leader to be a member of its own clique. Without this condition a leader could belong to a different clique, and `Cleanse` (below) could merge two cliques when collapsing. Since the leader is a member, the first disjunct already ensures that every member is adjacent to the leader. The predicate `E` is not defined in the ASP fragment: it receives edges from the graph `G` via `ClingoFact`.

Third, Logica takes the result back. The `Cleanse` function maps each conspirator to its clique leader, producing a smaller graph `CleansedG`. PageRank is recomputed on this graph via a functor: `PRC := PR(G: CleansedG)` reuses the original PageRank definition with a different input graph, without duplicating any code. The final `Ranks` predicate compares rankings before and after, revealing how influence redistributes when the conspiracies are neutralized.

Figure 6 shows the result of running this program on a larger randomly generated graph. Both graphs are drawn in a single layout: the original one, where cliques (dense subgraphs) are clearly visible, and the collapsed one, in which each clique is replaced by a single leader node. Dashed lines indicate which original nodes map to which leader. Logica includes built-in graph visualization, making it easy to inspect results interactively.

7. Related Work

Several systems have explored the boundary between ASP and external computation. DLVDB [28] integrates the DLV solver with relational databases, allowing ASP rules to query database tables directly, with a focus on grounding efficiency. HEX programs [10] extend ASP with external atoms that can invoke arbitrary computations during solving. Our integration works in the opposite direction: rather

```
# Given a symmetric graph G(x, y).

# PageRank (as defined in Section 2).
Node(x) distinct :- G(a, b), x in [a, b];
D(x) += 1 :- G(x, y);
PR(x) += ResetP() / NumNodes() :- Node(x);
PR(y) += PR(x) * WalkP() / (D(x) + 1) :- G(x, y);
PR(x) += PR(x) * WalkP() / (D(x) + 1);

##### ASP: maximum clique packing #####
N(x) :- x in [a, b], E(a, b);
Clique(x, c) couldbe :- N(x), N(c);
TooSmall() cantbe :- Clique(x, c), Count{y :- Clique(y, c)} < 4;
NotClique() cantbe :-
     Clique(x, c), Clique(y, c), x != y, ~E(x, y)
  |  Clique(x, c1), Clique(x, c2), c1 != c2
  |  Clique(x, c), ~Clique(c, c);
ConspiratorsCount(n) :- n = Count{x :- Clique(x, c)};
Paranoia() = Max{n :- ConspiratorsCount(n)} shouldbe;

# Invoke Clingo, passing graph edges as facts.
CliquePredicates() = ["Clique", "N", "TooSmall",
  "NotClique", "E", "Paranoia", "ConspiratorsCount"];
Models() = Clingo(CliquePredicates(), List{ClingoFact("E", [a, b]) :- G(a, b)});

# Extract the conspiracies from the optimal model.
BestConspiracy(member:, leader:) :-
  ExtractClingoCall(member, leader, predicate: "Clique", model_id:) = Models(),
  model_id = Max{m.model_id :- m in Models()};

# Cleanse: collapse each clique into its leader.
Cleanse(member) = leader :- BestConspiracy(member:, leader:);
Cleanse(x) = x :- Node(x), ~BestConspiracy(member: x);
CleansedG(cx, cy) distinct :- G(x, y), cx = Cleanse(x), cy = Cleanse(y), cx != cy;

# Recompute PageRank on the cleansed graph.
PRC := PR(G: CleansedG);

# Compare rankings before and after.
Ranks(node: x, original: PR(x), cleansed: PRC(x), ratio: PRC(x) / PR(x));
```

Figure 5: Conspiracy detection: finding and neutralizing cliques.

than extending ASP to access external data, we extend a SQL-based language to access ASP. The three operators are native Logica syntax, requiring no external code. Architecturally, HEX and our work are duals: HEX lets ASP call out to external computations; here, SQL calls into ASP.

Several extensions to ASP solvers address reasoning tasks beyond standard answer set computation. The asprin framework [7] provides a general mechanism for expressing preferences among answer sets, supporting both qualitative and quantitative criteria. Our `shouldbe` operator covers a narrower slice of this space, mapping to Clingo's `#maximize` and `#minimize`. Clingo's multi-shot solving [13] allows a Python control loop to ground and solve program parts incrementally. Our parameterized `Models(attack)` pattern in Section 5 achieves a similar effect, with DuckDB's per-row evaluation of the `Clingo()` function triggering independent solver invocations automatically, though without multi-shot's incremental grounding and retained solver state. However, the orchestration remains declarative, i.e., the programmer writes a logical rule, not an imperative loop. Alviano et al. [2] address subset-minimal answer set enumeration within the solver, computing minimal models without enumerating all answer sets first. Our strategic companies example (Section 4) instead filters for minimality on the SQL side; integrating solver-side minimality could reduce the number of answer sets that cross the

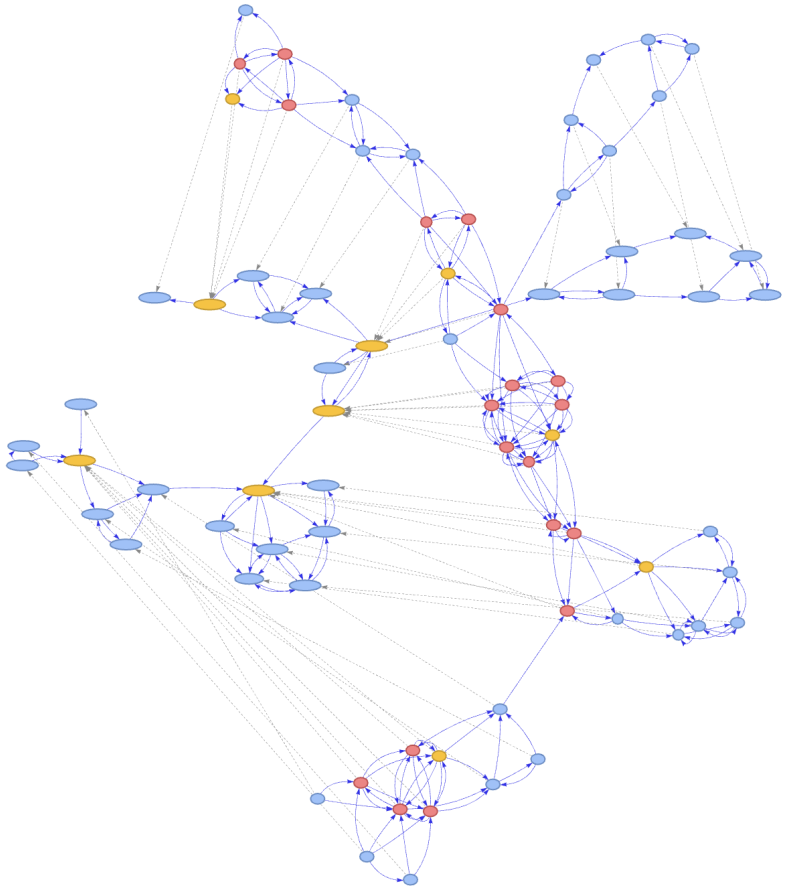


Figure 6: Conspiracy detection on a randomly generated graph. The original graph and the collapsed one are drawn in a single layout, keeping node colors in both copies: clique members in red, leaders in gold, regular nodes in blue. Dashed lines show the membership mapping.

SQL/ASP boundary.

A broader line of work avoids enumerating all answer sets: faceted answer set navigation [11, 21] interactively restricts the solution space via atoms that hold in some but not all answer sets; other techniques compute *diverse* [5] or *representative* [6] answer sets, or count answer sets under assumptions [12]. We instead materialize all answer sets, which keeps the analytical queries simple but limits scalability. Closest to our pipelines is ASP Chef [1], which chains ASP solving with filtering, aggregation, and visualization; in Logica such steps are ordinary rules in the same program.

ASP with Quantifiers (ASP(Q)) [3] extends ASP with explicit quantifiers over answer sets, enabling direct modeling of problems across the entire polynomial hierarchy, with the programmer structuring the alternation explicitly and the solver compiling to QBF. Our integration reaches similar problems by a different mechanism: the alternation emerges from combining Clingo's existential search with Logica's universal analysis over all solutions. The trade-off is that ASP(Q) keeps the whole computation within logic programming, while we use SQL for the analytical components. IDP-Z3 [9] pursues a similar "one language, multiple inferences" vision, where a single FO(·) knowledge base supports model expansion, propagation, and optimization. We share this ambition but start from Logica/Datalog rather than first-order logic, and delegate search to an external solver rather than building inference into the engine. Picat [29] integrates logic programming with constraint solving, SAT, and planning in one multi-paradigm language.

Non-deterministic choice has a long history in Datalog. The `choice` construct originates with Krishnamurthy and Naqvi [18]; Saccà and Zaniolo [22] and Giannotti et al. [15] refined it and showed that stable-model semantics provides a unified basis for negation and non-determinism. That con-

struct should be distinguished from `couldbe`: Datalog's `choice` enforces a functional dependency non-deterministically, selecting exactly one value per key, whereas `couldbe` compiles to an *unconstrained* Clingo choice rule and so admits an arbitrary subset. Soufflé's choice domain [16] follows the functional-dependency reading, useful for worklist algorithms but semantically different from ASP's subset-minimal answer sets. Beyond choice, several Datalog engines have pushed the classical language in other directions. Soufflé [23] focuses on high-performance deterministic evaluation, while Vadalog [4] (warded Datalog$^{\pm}$, evaluated with the warded chase) and Nemo [17] (existential rules with chase-based reasoning) support value invention through existential quantification. Our work extends Logica with ASP operators, adding non-deterministic search rather than existentials or raw performance.

8. Conclusion

We have presented an integration of ASP into Logica that combines the combinatorial search and optimization capabilities of Clingo with the data analysis strengths of SQL-based evaluation, within a unified logic programming framework. Unlike ad hoc approaches that stitch together separate languages and tools (e.g., Python scripts coordinating SQL queries and ASP solver calls), Logica and ASP share a common declarative foundation, allowing the programmer to express both search and analysis in a single, coherent rule-based program. Three operators, `couldbe`, `cantbe`, and `shouldbe`, extend Logica's syntax with ASP capabilities while deterministic rules continue to compile to SQL as before. The integration is implemented as user-defined functions registered with DuckDB, with no changes to the SQL engine itself.

We also showed the integration on three examples that each illustrate a distinct integration pattern. The strategic companies example uses Clingo as a *world explorer*, generating all solutions for Logica to compare. The adversarial coloring example *parameterizes* Clingo calls, running the solver once per scenario with Logica controlling the outer loop. The conspiracy example follows a *pipeline* where Logica analyzes, Clingo searches, and then Logica analyzes again.

Several promising directions remain as future work. ASP support is currently limited to DuckDB, and extending the integration to other backend engines would broaden its applicability. An automatic detection mode could complement the programmer's explicit choice of ASP predicates by inferring the non-deterministic fragment from the operators, simplifying common cases. Solver invocations are currently independent, and each materializes all its answer sets before SQL evaluation resumes. Support for Clingo's multi-shot solving [13] would allow shared grounding across invocations, improving performance for parameterized problems like the adversarial coloring example. Streaming answer sets one at a time instead of materializing them would reduce the memory cost of analyses that do not need all answer sets at once, such as the cautious test of Section 4.

Declaration on Generative AI

During the preparation of this work, the authors used Claude Code by Anthropic in an assistive capacity, to review, reword, and format text and citations. All output was reviewed and verified by the authors, who take full responsibility for the content. All references were checked against their original sources.

References

[1] Mario Alviano, Davide Cirimele, and Luis Angel Rodriguez Reiners. Introducing ASP recipes and ASP chef. In *Proceedings of the 16th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2023)*, volume 3437 of *CEUR Workshop Proceedings*, 2023. URL https://ceur-ws.org/Vol-3437/paper4ASPOCP.pdf.

[2] Mario Alviano, Carmine Dodaro, Salvatore Fiorentino, Alessandro Previti, and Francesco Ricca. ASP and subset minimality: Enumeration, cautious reasoning and MUSes. *Artificial Intelligence*, 320:103931, 2023. doi:10.1016/j.artint.2023.103931.

[3] Giovanni Amendola, Bernardo Cuteri, Francesco Ricca, and Mirek Truszczynski. Solving problems in the polynomial hierarchy with ASP(Q). In *Proceedings of the 16th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR 2022)*, volume 13416 of *LNAI*, pages 373–386. Springer, 2022. doi:10.1007/978-3-031-15707-3_29.

[4] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. The Vadalog system: Datalog-based reasoning for knowledge graphs. *Proceedings of the VLDB Endowment*, 11(9):975–987, 2018. doi:10.14778/3213880.3213888. URL https://www.vldb.org/pvldb/vol11/p975-bellomarini.pdf.

[5] Elisa Böhl and Sarah Alice Gaggl. Tunas—fishing for diverse answer sets: A multi-shot trade up strategy. In *Proceedings of the 16th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR 2022)*, volume 13416 of *LNAI*, pages 89–102. Springer, 2022. doi:10.1007/978-3-031-15707-3_8.

[6] Elisa Böhl, Sarah Alice Gaggl, and Dominik Rusovac. Representative answer sets: Collecting something of everything. In *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI 2023)*, pages 271–278. IOS Press, 2023. doi:10.3233/FAIA230280. URL https://dblp.org/rec/conf/ecai/BohlGR23.

[7] Gerhard Brewka, James P. Delgrande, Javier Romero, and Torsten Schaub. asprin: Customizing answer set preferences without a headache. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 1467–1474. AAAI Press, 2015. doi:10.1609/aaai.v29i1.9398.

[8] Marco Cadoli, Thomas Eiter, and Georg Gottlob. Default logic as a query language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, 1997. doi:10.1109/69.599933.

[9] Pierre Carbonnelle, Simon Vandevelde, Joost Vennekens, and Marc Denecker. Interactive configurator with FO(·) and IDP-Z3. *CoRR*, abs/2202.00343, 2022. doi:10.48550/arXiv.2202.00343. arXiv:2202.00343.

[10] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 90–96, 2005. URL https://www.ijcai.org/Proceedings/05/Papers/1353.pdf.

[11] Johannes K. Fichte, Sarah Alice Gaggl, and Dominik Rusovac. Rushing and strolling among answer sets—navigation made easy. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI 2022)*, volume 36, pages 5651–5659. AAAI Press, 2022. doi:10.1609/aaai.v36i5.20506.

[12] Johannes K. Fichte, Sarah Alice Gaggl, Markus Hecher, and Dominik Rusovac. IASCAR: Incremental answer set counting by anytime refinement. *Theory and Practice of Logic Programming*, 24(3):505–532, 2024. doi:10.1017/S1471068424000036.

[13] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with Clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019. doi:10.1017/S1471068418000054.

[14] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming (ICLP/SLP 1988)*, pages 1070–1080. MIT Press, 1988. URL https://dblp.org/rec/conf/iclp/GelfondL88.

[15] Fosca Giannotti, Dino Pedreschi, Domenico Saccà, and Carlo Zaniolo. Non-determinism in deductive databases. In *Deductive and Object-Oriented Databases (DOOD)*, volume 566 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1991. doi:10.1007/3-540-55015-1_7.

[16] Xiaowen Hu, Joshua Karp, David Zhao, Abdul Zreika, Xi Wu, and Bernhard Scholz. The choice construct in the Soufflé language. In *Proceedings of the 19th Asian Symposium on Programming Languages and Systems (APLAS 2021)*, volume 13008 of *LNCS*, pages 163–181. Springer, 2021. doi:10.1007/978-3-030-89051-3_10.

[17] 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)*, volume 385 of *EPTCS*, pages 333–335, 2023. doi:10.4204/EPTCS.385.35.

[18] Ravi Krishnamurthy and Shamim Naqvi. Non-deterministic choice in Datalog. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, pages 416–424. Morgan Kaufmann,

1988. doi:10.1016/B978-1-4832-1313-2.50038-X.

[19] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006. doi:10.1145/1149114.1149117. URL https://arxiv.org/abs/cs/0211004.

[20] Mark Raasveldt and Hannes Mühleisen. DuckDB: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 1981–1984, 2019. doi:10.1145/3299869.3320212. URL https://mytherin.github.io/papers/2019-duckdbdemo.pdf.

[21] Dominik Rusovac, Markus Hecher, Martin Gebser, Sarah Alice Gaggl, and Johannes K. Fichte. Navigating and querying answer sets: How hard is it really and why? In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning (KR 2024)*, 2024. doi:10.24963/kr.2024/60.

[22] Domenico Saccà and Carlo Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 205–217. ACM, 1990. doi:10.1145/298514.298572. URL https://dblp.org/rec/conf/pods/SaccaZ90.

[23] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, pages 196–206. ACM, 2016. doi:10.1145/2892208.2892226. URL https://souffle-lang.github.io/pdf/cc.pdf.

[24] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002. doi:10.1016/S0004-3702(02)00187-X.

[25] Evgeny Skvortsov, Yilin Xia, Shawn Bowers, and Bertram Ludäscher. The Logica system: Elevating SQL databases to declarative data science engines. In *Proceedings of the 5th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0 2024)*, volume 3801 of *CEUR Workshop Proceedings*, pages 69–73, 2024. URL https://ceur-ws.org/Vol-3801/short5.pdf.

[26] Evgeny Skvortsov, Yilin Xia, and Bertram Ludäscher. Logica: Declarative data science for mere mortals. In *Proceedings of the 27th International Conference on Extending Database Technology (EDBT 2024)*, pages 842–845, 2024. doi:10.48786/edbt.2024.84.

[27] Evgeny Skvortsov, Yilin Xia, Ojaswa Garg, Shawn Bowers, and Bertram Ludäscher. Diamonds are forever: Stabilization semantics for unrestricted aggregation and recursion in Logica. *Theory and Practice of Logic Programming*, 2026. To appear. Presented at the International Conference on Logic Programming (ICLP 2026).

[28] Giorgio Terracina, Nicola Leone, Vincenzino Lio, and Claudio Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming*, 8(2):129–165, 2008. doi:10.1017/S1471068407003158.

[29] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat*. Springer, 2015. doi:10.1007/978-3-319-25883-6.