FlowLog: Re-thinking Datalog for Fast and Extensible Static Analysis

Zhenghong Yu[1], Hangdong Zhao[2], Wanzhu Hou[1], Paraschos Koutris[1]
[1]University of Wisconsin-Madison [2]Microsoft Gray Systems Lab

Abstract

Datalog is widely used to build static analyzers, yet existing engines often force a tradeoff between efficiency and extensibility. In practice, static analyses are not run once and forgotten: users edit facts, tune rules, diagnose bottlenecks, and often need semantics beyond standard Datalog, leaving these tasks to ad hoc tooling or invasive engine rewrites.

We demonstrate FlowLog, a Datalog compiler that turns Soufflé-style programs into Differential Dataflow executables for efficient and extensible static analysis. Across 24 benchmarks derived from real-world workloads, FlowLog consistently outperforms state-of-the-art engines in runtime while remaining memory-efficient and scaling better.

The demonstration walks attendees through a DOOP points-to analysis. They *run* it, switching the same program from one-shot to incremental evaluation that retracts a fact and updates results in milliseconds; *tune* it, inspecting per-operator costs in a browser-based profiler and repairing a bad join order; and *extend* it with a k-core example that uses semantics beyond Datalog.

CCS Concepts

• **Software and its engineering** → **Automated static analysis**; **Compilers**.

Keywords

Datalog, static analysis tools, efficient execution, extensible systems, incremental computation, profiling

1 Introduction

Datalog is a powerful declarative language for static analysis. Users describe relations and rules; the engine handles joins, propagation, and fixpoint iteration. This abstraction has been widely applied to many workloads such as points-to analysis, call-graph construction, borrow checking, and binary disassembly [3, 6, 12].

This success has inspired the development of Datalog engines. Systems such as Soufflé [15], Flix [10], Flan [1], and Ascent [14] achieve high performance through domain-specific optimizations. While effective in their target settings, these systems tend to sacrifice extensibility: adding incremental maintenance or new semantics typically requires substantial system changes, and scaling to larger workloads frequently demands extensive engineering. Other systems avoid building from scratch by layering Datalog on top of existing databases or dataflow frameworks. RecStep [5] compiles Datalog into SQL and evaluates it one iteration at a time, but switching to the database on each iteration adds high synchronization overhead. DDlog [13] compiles Datalog directly into Differential Dataflow [11] (DD), yet empirical studies report substantial memory overhead, sometimes orders of magnitude higher than alternatives [7, 9].

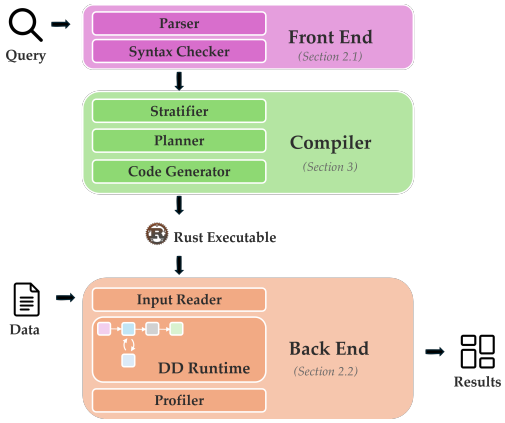


Figure 1: FlowLog architecture.

FlowLog [19] addresses this tradeoff with a new design that pursues efficiency and extensibility together. For efficiency, it builds on DD's off-the-shelf streaming operators as execution primitives. For flexibility, it introduces a relational intermediate representation (IR) that separates recursive control from each rule's logical plan, opening a dedicated layer for Datalog-aware rewrites that improve one-shot performance while enabling extensions such as incremental maintenance, profiling, and extended semantics.

This paper presents a tool demonstration of FlowLog as a user-facing artifact. Specifically, it makes the following contributions:

(1) **Tool overview with end-to-end usage.** Sec. 2 presents FlowLog's system overview and shows how users compile Soufflé-style Datalog programs into standalone executables for parallel one-shot execution.
(2) **Efficiency validation.** Sec. 3 evaluates FlowLog on 24 real-world static-analysis workloads, including Polonius-style borrow checking and DOOP-based points-to analysis, and compares it with state-of-the-art Datalog engines.
(3) **Extensible workflow.** Sec. 4 demonstrates incremental maintenance, profiling, and extended semantics through runnable examples that attendees can modify and explore.

The intended users are *software analysis researchers* evaluating new declarative analyses, *compiler and verification tool builders* who embed recursive analyses into larger tools, and *engineers* who operate deployed Datalog analyses and tune their performance.

A technical paper describes FlowLog's compiler design, optimization techniques, and comprehensive evaluation in detail [19].

2 Tool Overview and Usage

As shown in Fig. 1, FlowLog proceeds in three stages: ❶ **front-end**, which accepts an input Datalog program in Soufflé-style [15] syntax; ❷ **compiler**, which stratifies the validated program, plans

each rule, and generates an executable binary; and ❸ **back-end**, which reads the input data, runs the generated program on top of DD to produce the query output, and lets the profiler expose runtime metrics collected during execution.

2.1 Front-end

FlowLog's front-end accepts Datalog programs in a Soufflé-style syntax, with extensions such as richer types and recursive aggregation. After syntax and type checking, the front-end passes the validated program to the compiler.

We use the DOOP points-to analysis as the running example for compilation, execution, and profiling.

Example 2.1 (DOOP points-to). The program below shows part of the core logic of the DOOP points-to analysis for Java programs. For brevity, we elide many declarations and rules; the full program is included in the artifact.

```
// Input (EDB) relations extracted from Java bytecode
// Methods reachable from the program entry points
.decl Reachable(method: str)
// A field load in method inmethod: to = base.sig
.decl LoadInstanceField(base: str, sig: str, to: str, inmethod: str)
// ... other input relations omitted

// Derived (IDB) relations
// Variable var may point to object heap
.decl VarPointsTo(heap: str, var: str)
// Field sig of object baseHeap may point to object heap
.decl InstanceFieldPointsTo(heap: str, sig: str, baseHeap: str)
// ... other derived relations omitted

// Propagate points-to through a field load
VarPointsTo(heap, to) :-
  Reachable(inmethod),
  LoadInstanceField(base, sig, to, inmethod),
  VarPointsTo(baseheap, base),
  InstanceFieldPointsTo(heap, sig, baseheap).
// bad join order for the profiler demo: .plan (1, 3, 4, 2)
// ... other rules omitted

.output VarPointsTo
```

2.2 Compiler

FlowLog compiles a validated Datalog program into a standalone executable. The same source program can be compiled for one-shot execution, incremental maintenance, or profiling. Table 1 lists the main options; the command below compiles Example 2.1 in one-shot mode:

```
$ flowlog doop.dl -o doop-one-shot -F ./facts -D ./out
```

Table 1: FlowLog compile options.

Option	Description
`-D <dir>`	output result directory
`-F <dir>`	input fact directory
`-mode <m>`	execution mode; one-shot (default) or incremental
`-o <path>`	output binary
`-P`	enable profiling

Internally, the compiler stratifies the program and lowers each rule into a relational intermediate representation (IR). The IR is a tree of operators: leaves are input relations, and edges carry tuple schemas. Figure 2 shows the IR for the rule in Example 2.1. A `SemiJoin` first restricts `LoadInstanceField` to methods in `Reachable`, and two `Join` operators then combine the result with `VarPointsTo` and `InstanceFieldPointsTo` into new `VarPointsTo` tuples. These tuples feed back into the recursion along the dashed edge.

Join order largely determines intermediate result sizes, so the planner applies several Datalog-aware optimizations to shrink them and make execution less sensitive to join order: ❶ *pushdown*, which moves projections, filters, and semi/anti-joins as early as possible; ❷ *sideways information passing*, a Yannakakis-style semijoin reduction [17] that pre-filters atoms before the actual joins; and ❸ *subplan sharing*, which reuses repeated indexes across the program.

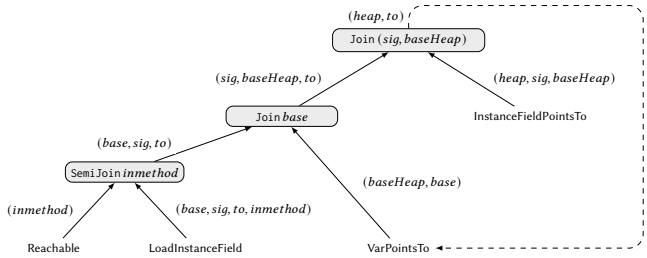


Figure 2: Optimized IR for the rule in Example 2.1.

The code generator then emits a DD dataflow in Rust from the optimized IR and builds the standalone executable. Each IR operator maps to one or more DD operators, and FlowLog relies on DD to run the program incrementally and in parallel to a fixpoint. The generated code also includes profiling instrumentation, discussed in Sec. 4.

2.3 Back-end

The back end loads the EDB facts, executes the binary on the DD runtime to a fixpoint, and derives the IDB results. The resulting binary can then be run with:

```
$ ./doop-one-shot -w 32    // run in parallel with 32 workers
```

DD is a streaming dataflow library. Its relations are collections of tuple updates tagged with logical time and an integer multiplicity, so operators process changes rather than whole relations. This yields efficient semi-naive evaluation: recursive iterations process only newly derived tuples, run in parallel across workers, and keep indexed state in arrangements that the profiler can inspect.

3 Efficiency Evaluation

Workloads. We evaluate on two Datalog static-analysis workloads derived from real analyses and datasets. *DOOP* [3] is a Java points-to analysis; we run a core, string-keyed port of it on real-world Java programs from the DaCapo suite (e.g., `tomcat`, `eclipse`). *Polonius* [12] is the Rust borrow checker; we run a simplified, integer-keyed version on Rust crates (e.g., `wgpu`, `clap-rs`).

Competing Engines. We compare against several state-of-the-art Datalog engines: (1) **Soufflé** (compiled) with its mature optimizer [2, 7, 8, 16]; (2) **DDlog** [13], which shares the same DD backend as FlowLog but lacks its relational IR and optimizations, making it a fair baseline. We also report results from (3) **Ascent** [14], a Datalog embedded in Rust through compile-time macros. We omit

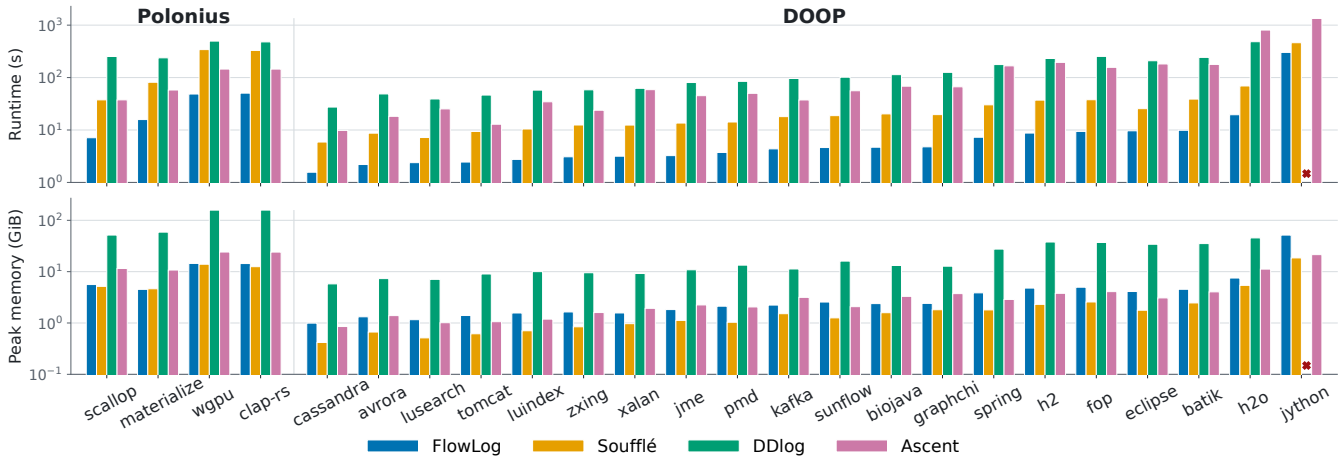


Figure 3: Runtimes (top) and peak memory (bottom) of FlowLog, Soufflé, DDlog, and Ascent on Polonius borrow-checking (left) and DOOP points-to (right) workloads for 32 threads (log scale; lower is better). × indicates an out-of-memory failure. FlowLog is the fastest on every benchmark in our study and uses far less memory than DDlog.

other Datalog systems because they are either not open source or lack support for features our workloads require. Reported runtimes exclude compilation time. During timing runs, programs emit only output cardinalities to avoid output-I/O overhead. We separately validated that all engines produce the same results on each benchmark.

Environment Setup. The benchmark artifact records the engine versions and scripts used for runtime, memory, and scalability measurements.[1] Experiments run on a CloudLab virtual machine [4] with two AMD EPYC 7543 32-core processors, hyper-threading enabled, and 256 GB RAM.

3.1 Results Summary

Runtime. Figure 3 (top) reports execution time on a log scale. FlowLog is the fastest on all 24 benchmarks. On DOOP, its median speedup is 3.9× over Soufflé (1.5–4.3×), 22× over DDlog (16–27×), and 14× over Ascent (4.4–41×); on Polonius, it is 5.9× over Soufflé (5.1–7.0×), 13× over DDlog (9.5–35×), and 3.3× over Ascent (2.9–5.3×). FlowLog's IR-level optimizations (Section 2) shrink and reuse intermediate state, and DD's asynchronous runtime keeps every operator active across the iterations.

Memory. Figure 3 (bottom) reports peak memory. FlowLog is far more memory-efficient than DDlog, which uses a median of 6.3× more (5–13×), because FlowLog's subplan sharing avoids the redundant intermediate state that DDlog materializes. Against Soufflé, FlowLog uses about 2× more memory on DOOP (1.3–2.8×) and stays on par on Polonius (~ 1.1×). This is a deliberate trade-off: FlowLog materializes all of its indexes as DD arrangements, which speeds execution and, unlike Soufflé's one-shot-only design, leaves it ready to maintain results incrementally (Section 4).

Scalability. Static analyses converge through a fixpoint of many small iterations, each doing little work. This makes parallelism hard: coordinating threads and exchanging data at every iteration can cost more than the iteration computes. Figure 4 shows the effect as we scale to 32 threads, where Soufflé, DDlog, and Ascent barely move past a single thread on the highly iterative Polonius workload. FlowLog avoids this because it compiles to DD, whose asynchronous, data-parallel runtime keeps every worker busy across these lightweight iterations rather than stalling at each one, so it scales the most consistently, reaching 15× on Polonius and 8× on DOOP.

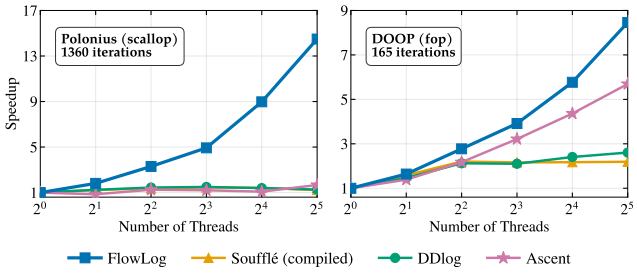


Figure 4: Scalability of all competing systems on two program-data pairs; each subplot shows speedups relative to the single-threaded run, up to 2^5 threads. Legends give the program-data pair and iterations to converge.

4 Extensible Workflow

FlowLog's relational IR separates rule-level logic from the physical DD runtime, enabling extensions to execution, instrumentation, and semantics without re-engineering the core.

4.1 Incremental Maintenance

To run incrementally, users add `-mode datalog-inc` (Table 1) when compiling; this produces an executable with an interactive shell (Table 2).

```
$ flowlog doop.dl -o doop-inc -F ./facts -D ./out --mode datalog-inc
```

We compile Example 2.1 in incremental mode and stage updates at two timestamps on the `tomcat` input: at `[t=0]` we load the base facts; at `[t=1]` we delete one `LoadInstanceField` fact, and

[1] https://github.com/flowlog-rs/flowlog-bench

Table 2: Interactive shell commands for incremental mode.

Command	Description
`begin`	Begin an update round.
`put rel tuple diff`	Stage one tuple update.
`file rel path diff`	Stage updates from a file.
`commit`	Apply staged updates at one logical time.

FlowLog incrementally retracts the single `VarPointsTo` fact that depended on it.

```
// t=0: load base facts and run the analysis
[t=0] » begin
[t=0] » file LoadInstanceField load.csv +1
// ... load other EDB relations
[t=0] » commit
// tuples omitted; report only the VarPointsTo size
[t=0] VarPointsTo size=4,986,481
// t=1: remove one load from ParameterParser.chars
[t=1] » begin
[t=1] » put LoadInstanceField ParameterParser.chars ->
      getToken/$stack44 -1
[t=1] » commit
[t=1] VarPointsTo data=(char[]@String.toCharArray, getToken/$stack44)
      diff=-1
```

Applying this retraction takes about 7 ms, versus 2.23 s for full recomputation (320× faster), because FlowLog propagates only the affected deltas.

4.2 Profiler-Guided Diagnosis

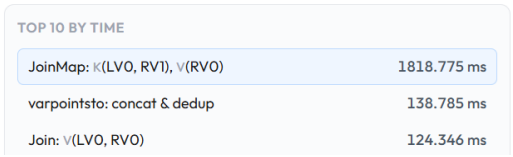
A poor join order can cripple a recursive rule, while choosing a good one from the rule text alone usually relies on expert intuition, trial and tuning. FlowLog's profiler and visualizer expose per-operator cost, so users can localize bottlenecks without reading the generated dataflow code. Users can set the order with a `.plan` directive over rule atoms. In Example 2.1, the three core atoms are `VarPointsTo` (V), `InstanceFieldPointsTo` (I), and `LoadInstanceField` (L); `Reachable` is always pushed down first as a semijoin (Section 2), and FlowLog's default plan starts the joins from L, which is a good order. To make the demonstration reproducible, we deliberately pin the bad order $(V \bowtie I) \bowtie L$ via the commented `.plan` directive. In normal use, users rarely know a good order in advance; they observe slow runs, then use the profiler and rule-level visualization to find expensive operators and compare plans.

Compiling with `-P` taps DD's hooks in the generated dataflow to record per-operator metrics, and running the binary writes them to a log directory. `flowlog-visualizer`[2] renders these logs into a standalone HTML report, which ranks operators and links them back to the corresponding rule plan (Figure 5). It also offers a memory top-10, per-operator metrics, and an interactive dataflow graph.

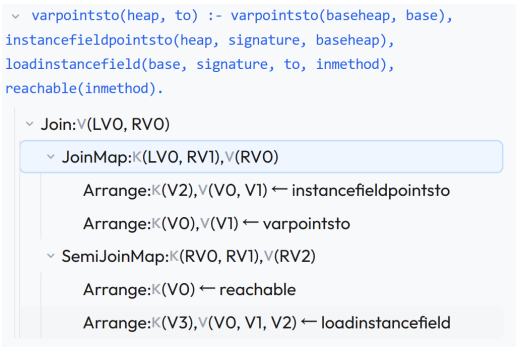
```
$ flowlog doop.dl -o doop-prof -F ./facts -D ./out -P
```

The profiler reports fused physical joins as `Join-Map` operators. A single `Join-Map` dominates the time ranking, indicating that the plan spends most of its time in one join. Selecting it highlights the matching node in the rule plan (Figure 5b), revealing the executed order $(V \bowtie I) \bowtie L$. The user closes the loop by deleting the `.plan` directive and restoring the default good order; recompiling cuts runtime from `6.5s` to `2.2s` on `tomcat` (`220s` to `9.8s` on `fop`).

[2]https://github.com/flowlog-rs/flowlog/tree/main/flowlog-visualizer



(a) Excerpt of the top-10 operators by time; one `Join-Map` dominates.



(b) Rule plan for Example 2.1.

Figure 5: FlowLog profiler views for the target rule under the poor join order.

4.3 Extended Semantics

Because FlowLog lowers programs to DD dataflows, it can support controlled extensions beyond standard Datalog, including branching, loop control, and non-monotonic updates. We illustrate the non-monotonic case with k-core (Example 4.1), which standard Datalog cannot express because derived facts are never retracted. In FlowLog, a `fixpoint` block re-runs its rules until stable, and `.iterative` marks relations allowed to shrink across rounds. The example keeps these semantics in the same compiled pipeline as ordinary Datalog rules; the same mechanism can also capture equality-saturation-style rewriting in the spirit of egglog [18]. Full details appear in the online tutorial.[3]

Example 4.1 (k-core computation). Computing the k-core repeatedly removes vertices whose current degree is below k until all remaining vertices have at least k neighbors.

```
// the block re-evaluates until it reaches a fixpoint
fixpoint {
    // .iterative marks relations that are non-monotonic
    .iterative active_edge
    .iterative degree
    active_edge(x, y) :- edge(x, y), !removed(x), !removed(y).
    degree(x, count(y)) :- active_edge(x, y).
    removed(x) :- degree(x, d), d < k.
}
```

5 Tool Availability

Tool: https://github.com/flowlog-rs/flowlog. **Docs:** https://www.flowlog-rs.com/tutorial/intro/. Docs cover installation, command-line use, example analyses, incremental updates, profiling, and extended semantics. **Video:** https://youtu.be/DNU1Uzt47MI. **Archive:** https://doi.org/10.5281/zenodo.20815411.

[3]https://www.flowlog-rs.com/tutorial/language/extended-semantics

References

[1] Supun Abeysinghe, Anxhelo Xhebraj, and Tiark Rompf. 2024. Flan: An Expressive and Efficient Datalog Compiler for Program Analysis. *Proc. ACM Program. Lang.* 8, POPL (2024), 2577–2609. doi:10.1145/3632928

[2] Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotic, and Bernhard Scholz. 2022. Building a Join Optimizer for Soufflé. In *LOPSTR (Lecture Notes in Computer Science, Vol. 13474)*. Springer, 83–102.

[3] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. doi:10.1145/1640089.1640108

[4] CloudLab 2018. https://www.cloudlab.us/.

[5] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. *Proc. VLDB Endow.* 12, 6 (2019), 695–708.

[6] Antonio Flores-Montoya and Eric M. Schulte. 2020. Datalog Disassembly. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1075–1092. https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[7] Xiaowen Hu, David Zhao, Herbert Jordan, and Bernhard Scholz. 2021. An efficient interpreter for Datalog by de-specializing relations. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 681–695. doi:10.1145/3453483.3454070

[8] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. Brie: A Specialized Trie for Concurrent Datalog. In *PMAM@PPoPP*. ACM, 31–40.

[9] Yuanbo Li, Kris Satya, and Qirun Zhang. 2022. Efficient algorithms for dynamic bidirected Dyck-reachability. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. doi:10.1145/3498724

[10] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 194–208. doi:10.1145/2908080.2908096

[11] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[12] Rust Project Developers. 2018. Polonius: The Rust Borrow Checker. https://github.com/rust-lang/polonius. Accessed June 2026.

[13] Leonid Ryzhyk and Mihai Budiu. 2019. DDlog: A Language for Programming Incremental Computations. https://github.com/vmware/differential-datalog.

[14] Arash Sahebolamri, Thomas Gilray, and Kristopher K. Micinski. 2022. Seamless deductive inference via macros. In *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, Bernhard Egger and Aaron Smith (Eds.). ACM, 77–88. doi:10.1145/3497776.3517779

[15] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. 196–206.

[16] Pavle Subotic, Herbert Jordan, Lijun Chang, Alan D. Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. *Proc. VLDB Endow.* 12, 2 (2018), 141–153.

[17] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. IEEE Computer Society, 82–94.

[18] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI (2023), 468–492. doi:10.1145/3591239

[19] Hangdong Zhao, Zhenghong Yu, Srinag Rao, Simon Frisk, Zhiwei Fan, and Paraschos Koutris. 2025. FlowLog: Efficient and Extensible Datalog via Incrementality. *Proceedings of the VLDB Endowment* 19, 3 (2025), 361–374.