

# FlowLog: Efficient and Extensible Datalog via Incrementality

Hangdong Zhao

Microsoft Gray Systems Lab

hangdongzhao@microsoft.com

Simon Frisk

University of Wisconsin, Madison

simon.frisk@wisc.edu

Zhenghong Yu

University of Wisconsin, Madison

zyu379@wisc.edu

Zhiwei Fan

Meta Platforms Inc.

zhiweifan@meta.com

Srinag Rao

University of Wisconsin, Madison

srinskit@cs.wisc.edu

Paraschos Koutris

University of Wisconsin, Madison

paris@cs.wisc.edu

## ABSTRACT

Datalog-based languages are regaining popularity as a powerful abstraction for expressing recursive computations in domains such as program analysis and graph processing. However, existing systems often face a trade-off between efficiency and extensibility. Engines like Soufflé achieve high efficiency through domain-specific designs, but lack general-purpose flexibility. Others, like RecStep, offer modularity by layering Datalog on traditional databases, but struggle to integrate Datalog-specific optimizations.

This paper bridges this gap by presenting FlowLog, a new Datalog engine that uses an explicit relational IR per-rule to cleanly separate recursive control (e.g., semi-naïve execution) from each rule’s logical plan. This boundary lets us retain fine-grained, Datalog-aware optimizations at the logical layer, but also reuse off-the-shelf database primitives at execution. At the logical level (i.e. IR), we apply proven SQL optimizations, such as logic fusion and subplan reuse. To address high volatility in recursive workloads, we adopt a robustness-first approach that pairs a structural optimizer (avoiding worst-case joins) with sideways information passing (early filtering). Built atop Differential Dataflow—a mature framework for streaming analytics—FlowLog supports both batch and incremental Datalog and adds novel recursion-aware optimizations called Boolean (or algebraic) specialization. Our evaluation shows that FlowLog outperforms state-of-the-art Datalog engines and modern databases across a broad range of recursive workloads, achieving superior scalability while preserving a simple and extensible architecture.

### PVLDB Reference Format:

Hangdong Zhao, Zhenghong Yu, Srinag Rao, Simon Frisk, Zhiwei Fan, and Paraschos Koutris. FlowLog: Efficient and Extensible Datalog via Incrementality. PVLDB, 19(3): XXX-XXX, 2025.

doi:XX.XX/XXX.XX

### PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/hdz284/FlowLog>.

## 1 INTRODUCTION

The rapid expansion of data-intensive applications has underscored the need for query languages that are both simple and expressive.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.

doi:XX.XX/XXX.XX

As a declarative language, Datalog adds recursion to relational algebra in a concise syntax, making it especially well-suited for domains such as graph processing [14, 30], network monitoring [2], program analysis [49, 56], and distributed systems [10].

In recent years, academic advances and industry demands have spurred the development of Datalog engines. In program analysis, Datalog has proven effective for static analysis tasks such as bug detection and security checks, inspiring systems such as Soufflé [49], Flix [36], Flan [1] and Ascent [48], often with domain-specific optimizations [24, 54]. While successful in their target domains, these systems tend to sacrifice flexibility: adding incremental maintenance for continuous updates typically requires major system modifications [65], and scaling up/out to accommodate larger datasets frequently demands extensive engineering.

Other systems avoid building from scratch by layering Datalog on existing databases. A prominent example is RecStep [14], which compiles Datalog into SQL and executes it in QuickStep [44] one iteration at a time, inheriting its optimizer and parallelism. However, relying on a black-box DBMS complicates Datalog-specific optimizations, particularly those spanning multiple iterations, such as semi-naïve evaluation and index maintenance. While RecStep interjects between iterations to impose some of these optimizations, the back-and-forth control flow incurs non-negligible synchronization overhead. Similarly, DDlog [46] compiles Datalog directly into Differential Dataflow<sup>1</sup> (DD) programs. However, empirical studies report substantial memory overhead—often orders of magnitude higher than alternatives [22, 34, 38, 65].

Balancing flexibility and efficiency for Datalog remains an open challenge. This paper addresses this by pursuing a new design that unifies (i) efficient, off-the-shelf relational operators as execution primitives, and (ii) flexible, fine-grained optimization controls. To achieve the first goal, we reuse DD’s streaming operators as building blocks, positioning DDlog as a baseline that directly code-generates Datalog into lower-level DD without a distinct optimization phase. Achieving the second goal requires a novel design for Datalog optimization. There are two main reasons for this: (i) the primary bottleneck of DDlog is memory usage, and (ii) recursion weakens many conventional SQL techniques—e.g., cost-based planning [32] lacks reliable static statistics in recursive contexts—so existing systems (e.g., Soufflé, DDlog) fall back on ad hoc heuristics or manual performance tuning. To curb memory usage, we apply a suite of memory-focused rewrites on a relational intermediate representation (IR) that fully separates the rule’s logical plan from its physical

<sup>1</sup>Differential dataflow [41] programs chain up a set of DD’s streaming operators that continuously maintain states for efficient incremental computation as data evolves (see Sec. 2.3 for a formal introduction).

realization. Additionally, instead of the conventional cost-based planning, we opt for robustness-aware optimizations, which prioritizes avoiding worst-case scenarios (e.g., pathological join orders and large intermediate relations) over seeking a best query plan.

These pieces culminate in FlowLog: a high-performance Datalog system targeting both batch and incremental processing. Its simple architecture eases integration of novel optimizations (e.g. Boolean specialization, Sec. 8), rich semantics (e.g. recursive aggregations) and scale-out extensions (Sec. 9) with minimal system changes. In summary, this paper makes the following **contributions**:

- (1) **System Architecture.** Sec. 3 presents the new system design of FlowLog, centered around an IR that decouples every rule’s logical structure from its physical execution in DD. While this logical/physical split is long standard in modern SQL databases, it has been largely absent from existing Datalog engines, limiting systematic Datalog optimization.
- (2) **Optimization & Robustness.** FlowLog integrates a suite of IR-level (i.e. logical) optimizations, including proven SQL techniques such as logic fusion (Sec. 4) and subplan sharing (Sec. 7) to shrink DD’s state and memory footprint. Sec. 5 presents FlowLog’s optimizer that analyzes the join graph of each rule to avoid pathological join orders. Sec. 6 complements this by semijoin pre-filtering that stabilizes the execution of recursive workloads. Both techniques (Sec. 5-6) are geared towards mitigating the inherent high volatility and unpredictability we observed in Datalog workloads.
- (3) **Extensions.** Sec. 8-9 outline how FlowLog’s modular design supports (i) incremental maintenance, (ii) novel Datalog-aware optimizations (i.e., Boolean/algebraic specialization for recursive aggregation), and (iii) scale-out execution.
- (4) **Experiments.** Sec. 10 conducts extensive experiments on a broad set of benchmarks we have collected across recent literature and open-source projects, spanning multiple domains and heterogeneous workload characteristics. Our results show that FlowLog often substantially outperforms state-of-the-art Datalog engines and modern databases, in terms of latency, memory usage, and scalability.

**Related Work.** As [14, 25] pointed out, most Datalog engines are purpose-built for specific domains [1, 21, 22, 36, 48, 49, 51, 52, 55, 56]. Over time, they have introduced a range of optimizations and extensions tailored to Datalog [17, 18, 22, 27, 50, 60, 64, 67], such as incremental Datalog [46, 65]. We incorporate some of these techniques into FlowLog (e.g., index sharing [40, 54], unified IDB evaluation [14]), while others, such as magic sets [57], de-specialized relations [22], customized data structures [47] and worst-case optimal joins [1], remain promising future explorations. Notably, major gaps in the Datalog literature persist, particularly the lack of effective query planning and limited scalability in highly iterative workloads, both being critical challenges for large-scale Datalog applications [4, 13].

## 2 BACKGROUND

In this section, we provide a brief overview of standard Datalog, its evaluation, and common extensions.

### 2.1 Datalog Basics

A standard Datalog program [3] is a set of *rules*. A rule is an expression of the following form:

```
h :- p1, p2, ..., pk.
```

The terms  $h$ ,  $p_1, \dots, p_k$  are atoms, i.e., formulas of  $R(x, y, \dots)$ , where  $R$  is the atom (or relation) name and  $(x, y, \dots)$  is its variables (or attributes). The atom  $h$  is the *head* and the atoms  $p_1, \dots, p_k$  are the *body* of the rule. A rule can be interpreted as a logical implication: if  $p_1, \dots, p_k$  are true, then so is the head  $h$ . We assume that every attribute of  $h$  occurs in some  $p_i$ . The atoms of a Datalog program are of two types: IDB and EDB. An atom that represents an input relation is an EDB (extensional database); an IDB comprises a set of (base) facts/tuples and is never the head of a rule. A atom that represents a derived relation is an IDB (intensional database, in **bold** font); an IDB must appear in the head of at least one rule.

*Example 2.1.* Consider the following task over a directed graph: find all nodes that can reach a target via an even number of hops. We represent the graph as a binary EDB  $\text{edge}(x, y)$ , where  $(x, y)$  is a fact if there is an edge from node  $x$  to node  $y$ . We use another EDB  $\text{target}(x)$  as the unary relation containing the target node  $a$ . The task can then be expressed in Datalog as follows:

```
r1.  reach(x) :- target(x).
r2.  reach(x) :-  $\text{edge}(x, y)$ ,  $\text{edge}(y, z)$ , reach(z).
```

Here, the atom  $\text{reach}(x)$  is the IDB to output.  $r_1$  initializes the trivial case: the target node is reachable in zero hop. The second rule  $r_2$  is recursive and states that if there is a length-2 path from  $x$  to  $z$ , and  $z$  can reach the target using an even number of hops, then so can  $x$ .

**Dependency Graph and Stratification.** A *dependency graph* of a Datalog program is a directed graph where every rule is a node; there is an edge from rule  $r_1$  to  $r_2$  if the head of  $r_1$  appears in the body of  $r_2$ . A rule is *recursive* if it belongs to a directed cycle, and *non-recursive* otherwise. A *stratification* of a program is a partition of the rules into strata, where each stratum is the set of rules that belongs to the same strongly connected component of the dependency graph. The topological order of the strongly connected components defines the order among the strata. The dependency graph for Example 2.1 has two nodes  $r_1$  and  $r_2$ , and edges  $r_1 \rightarrow r_2$  and  $r_2 \rightarrow r_2$ . Thus, the program has strata,  $\{r_1\}$  and  $\{r_2\}$  in the topological order.

**Common Datalog Extensions.** To enrich Datalog for practical usage, we incorporate common syntactic extensions as [14]—constraints, stratified negations (where negated atoms are either EDB or an IDB from a lower stratum), and (possibly recursive) aggregations. For example, this allows finding two hops  $(x, z)$  in a graph that are:

```
 $\text{edge}(x, y)$ ,  $\text{edge}(y, z)$ ,  $x \neq z$  // not loops
 $\text{edge}(x, y)$ ,  $\text{edge}(y, z)$ ,  $\neg\text{edge}(x, z)$  // not one hop
```

or for each  $x$  and  $z$ , counting the number of possible two hops:

```
two_hops(x, z, COUNT(y)) :-  $\text{edge}(x, y)$ ,  $\text{edge}(y, z)$ .
```

### 2.2 Datalog Evaluation

The straightforward way to implement Datalog is via *naïve* evaluation. Starting with the set of all EDB facts, we iteratively apply every rule as a join query to derive new facts, adding them to the head IDBs until no new facts can be derived, i.e., a fixpoint is reached.

However, naïve evaluation is usually wasteful because each iteration executes rules on all historical data, leading to rediscovery of facts derived in previous iterations. Hence, modern Datalog engines use *semi-naïve* evaluation, which only uses new tuples from the last iteration to derive facts in the current iteration. A common practice is to further exploit stratification: each stratum gets evaluated in order, and the results are used as input for the next stratum. In Example 2.1, the first stratum simply inserts the target node  $t$  to  $\text{reach}(x)$ . Next, at each iteration  $i = 1, 2, \dots$ , we only consider the new facts derived in the  $(i - 1)$ -th iteration, denoted as  $\Delta\text{reach}^i$  where  $\Delta\text{reach}^0 = \{\text{a}\}$ , to populate new facts by computing the join  $\text{edge}(x, y) \bowtie \text{edge}(y, z) \bowtie \Delta\text{reach}^{(i-1)}$ .

### 2.3 Differential Dataflow

Differential Dataflow (or DD) [41] is a data-parallel programming model for large-scale incremental data processing. Its Rust implementation<sup>2</sup> compiles down to Timely Dataflow, a lower-level generic distributed streaming system introduced by [42].

**Collections.** DD abstracts a relation as a stream of rows, termed as a collection. A row in a collection is a triple  $(\text{data}, \text{time}, \text{diff})$ , where  $\text{data}$  is the raw tuple from the relation,  $\text{time}$  is the timestamp when the tuple is ingested, and  $\text{diff}$  is its multiplicity. The  $\text{diff}$  field is for DD to annotate duplications and track incremental changes (i.e.,  $+\delta$  represents an insertion of  $\delta$  copies of the tuple and  $-\delta$  represents a retraction of  $\delta$  copies). In Example 2.1, DD constructs corresponding input collections for  $\text{target}(x)$  and  $\text{edge}(x, y)$ , where the former is initialized as a single row  $(\text{a}, 0, 1)$  for the target  $\text{a}$ ; the latter is a collection of rows  $((\text{a}, \text{b}), 0, 1)$  for each edge  $(\text{a}, \text{b})$  of the graph. Here, 0 is the initial timestamp.

**Differential Operators.** DD uses a set of incremental operators such as `map`, `filter`, and `join`, that each imposes a relational operation on input collection(s) and outputs a collection. We can compose them to compute SQL queries and quickly respond to input changes. At its core, every differential operator maintains a minimal set of changes to the output when the input changes, and propagates these updates in a semi-naïve manner – only considering changes since the last timestamp. Let  $R(x, y)$  and  $S(x, z)$  be two collections. Suppose  $((\text{a}, \text{b}), 0, 4)$  and  $((\text{c}, \text{b}), 0, 1)$  are two rows of  $R(x, y)$  and  $((\text{a}, \text{d}), 0, 3)$  is a row of  $S(x, z)$ . A rule  $\text{T}(y, z) :- R(x, y), S(x, z)$  maps to a composition of `join` and `map` as follows:

```
R.join(&S).map(|t| (t.y, t.z));
```

Here, `join` emits a row  $((\text{a}, (\text{b}, \text{d})), 0, 12)$ , where the output data is formatted as  $(x, (y, z))$  (i.e. join keys followed by a grouping of values) and  $\text{diff}$  is the product of two input  $\text{diff}$ s, i.e.,  $4 \times 3 = 12$ . The downstream `map` projects to  $(y, z)$  and gets  $((\text{b}, \text{d}), 0, 12)$ . If there is an insertion of  $((\text{a}, \text{b}), 0, +2)$  in the collection, only a delta change  $((\text{b}, \text{d}), 0, +2 \times 3 = +6)$  will propagate through.

DD allows users to define custom operators (beyond standard SQL) without worrying about low-level incremental mechanisms. This makes DD a powerful backend for Datalog applications. A unique but essential operator is `iterate`, which repeatedly applies an enclosed DD closure to input collections. The following code snippet shows how to implement Example 2.1 as a DD program.

```
// (1) iterate starting from target
target.iterate(|reach| {
    // (2) derive new reach(x) from joins
    edge.map(|t| (t.y, t.x))
        .join(&edge)
        .map(|t| (t.z, t.x))
        .join(&reach)
        .map(|t| t.x)
    // (3) concat and dedup for convergence
    .concat(&reach)
    .distinct()
});
```

The `iterate` operator initializes `reach` as  $(\text{a}, (0, 0), 1)$ , i.e. target, and sets a series of nested timestamps  $(0, i)$ , where 0 is the outer timestamp and  $i = 0, 1, \dots$  is the iteration counter. Then, it repeatedly applies the inner DD logic as  $i$  increments—for each  $i$ , the `concat` operator adds new output into `reach` and `distinct` de-duplicates the results, i.e. for each row, `diff` maps to 1 if  $\text{diff} > 1$ . When no new rows are derived (i.e. fixpoint), `iterate` collects all results and returns the final `reach` to the outer scope/timestamp 0.

The inner logic is verbose but necessary. An idiosyncratic feature of DD is the use of `map` and `join`; `map` is not only a projection, but also a way to re-organize data into a key-value pair, e.g., the first `map` swaps  $(x, y)$  to  $(y, x)$ , and that designates  $y$  as the key and  $x$  as the value. This is because the `join` operator of DD requires its two input collections to be explicitly pre-indexed on the join keys, using an `arrange` operator described next.

**Arrangements.** An arrangement is an in-memory index for DD collections [40]. It can be considered as a sorted dictionary that allows efficient concurrent access. An arrangement indexes batches of historical changes, maintains them over time, and merges them into compact representations as appropriate (e.g., when a timestamp is advanced). The first `join` in the above code pre-arranges both operands by imposing an `arrange` operation internally for each and then uses a more primitive `join_core` operator to join on arrangements. Indeed, the first `join` is executed under the hood as

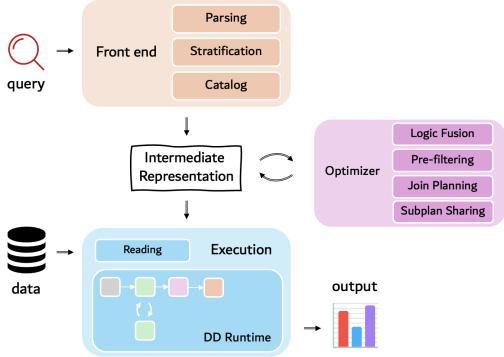
```
edge.map(|t| (t.y, t.x))
    .arrange() // k: (y), v1: (x)
    .join_core(
        &edge.arrange() // k: (y), v2: (z)
    ); // output schema k: (y), v3: (x, z)
```

## 3 SYSTEM DESIGN

This section presents the architecture of FlowLog, which builds on DD as its execution backend. Instead of compiling Datalog directly into a DD program, FlowLog first translates rules into an intermediate representation (IR) that captures the logical structure of the program, then lowers to DD. This allows us to reason about optimizations purely at the logical level while abstracting away low-level details such as recursive control, de-duplication, and incrementality, making it simpler to modularize and extend. The IR is akin to recent *grounding* approaches for Datalog [27, 66] and we incorporate these ideas when optimizing the IR (see Section 5).

**Overview.** The architecture of FlowLog has three components: the front-end, the optimizer, and the executor. The **front-end** parses the input `.dl` program (using Soufflé grammar), performs syntax checking, stratifies the rules, and creates a per-rule catalog with meta information such as the join graphs (formally defined in Section 5.2). The **optimizer** populates an IR from the catalog and

<sup>2</sup><https://github.com/TimelyDataflow/differential-dataflow>



**Figure 1: System Architecture of FlowLog**

applies optimizations. The **executor** renders the optimized IR to a DD program, reads the input data, and runs the iterative execution.

**Query Optimization (IR).** For every rule in the input Datalog program, FlowLog’s optimizer constructs an IR. The IR is a tree of logical transformations for that rule (i.e. a relational logical plan), e.g. Join, Map, and Filter. Leaf nodes are input tables and intermediate nodes are logical transformations. Edges denote the data flowing across these transformations, and edge labels indicate the underlying schema. We insist every Join’s inputs to align on its join keys so that DD’s physical join operator can consume it directly.

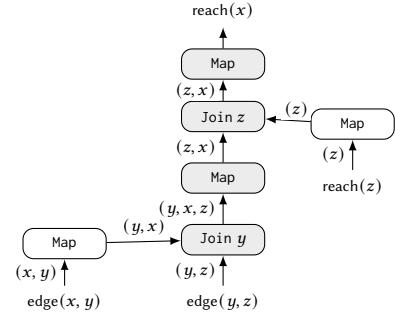
Figure 2a presents an IR for  $r_2$  (Example 2.1). The lowest Map of edge( $x, y$ ) assigns  $y$  as the join key ( $x$  as the value) for its parent Join, which binds both inputs to  $y$  and pushes the join result of schema  $(y, x, z)$  upstream; edge labels are omitted when the schema is obvious. The IR always reads like an ordinary SQL query plan, and thus easily accepts a suite of relational optimizations.

**Query Execution.** The executor takes a set of (optimized) IR (one for each input rule) and coalesces it into a global dataflow graph of (physical) differential operators, which are executed iteratively, strata by strata, until fixpoint. This execution taps into DD’s incremental and asynchronous nature, achieving efficiency and scalability out of the box. However, the incrementality asks every operator to maintain its state in memory. As exhibited by DDlog against systems such as Soufflé, the key overhead stems from these internal footprints, which can be prohibitive for large intermediate results. As such, Sec. 4-7 will each present an IR-level optimization (e.g. fusing transformations, re-using subplans) that are all geared towards minimizing intermediate output sizes, thus making FlowLog much more competitive than DDlog for large-scale recursive queries.

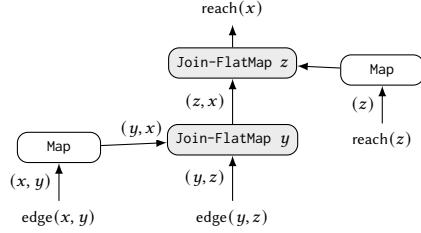
## 4 LOGIC FUSION

Logic fusion is a query optimization technique that merges multiple small and adjacent logic (of frequent occurrence) into a single one to reduce dispatches during interpretation. For FlowLog, logic fusion in addition eliminates unnecessary intermediate operator states. We describe two most effective fusion patterns for FlowLog’s IR and assume that they will always be applied in the rest of the paper.

**Consecutive Map and Filter.** We introduce a new FlatMap transformation into the IR that fuses consecutive Map and Filter (similar



**(a) Initial IR for  $r_2$ , shaded parts are consecutive Join and Map.**



**(b) An optimized IR by fusing Join and Map into Join-FlatMap.**

**Figure 2: Logic Fusion for  $r_2$  from Example 2.1**

fusions are applied to optimize incremental SQL<sup>3</sup>). A FlatMap mirrors a lower-level `flat_map` physical operator of DD that filters and projects tuples to the desired schema in one pass. For example, the following rule and code snippet find the neighbors of node  $a$ :

```
// neighbor(y) :- edge(x, y), x = a.
let neighbor = edge.flat_map(|t|
  if t.x == a { vec![(t.y)] } else { vec![] } );
```

**Join followed by Map/Filter.** A Join-FlatMap fuses a Join with subsequent Map(s) and Filter(s). It avoids materializing the full join output that is immediately projected or filtered. We will apply it as a default optimization for FlowLog’s IR. The initial IR of  $r_2$  (Figure 2a) is optimized by fusing every consecutive Join and Map. In the fused IR (Figure 2b), the lower Join-FlatMap directly emits  $(z, x)$  tuples; and the upper one now emits  $(x)$  tuples instead of  $(z, x)$ . At the executor, Join-FlatMap renders into a `join_core` physical DD operator, which works as the following pseudocode:

```
edge.join_core(&edge, |t|
  if some filters are passed // if any
    vec![(t.z, t.x)] // map to (z, x)
  else vec![] ); // filtered out
```

## 5 JOIN-PROJECT PLAN OPTIMIZATION

Traditional DBMS optimizers rely on approximate statistics to choose plans at the lowest cost [33]. However, finding optimal query plans for Datalog rules is much more strenuous as these statistics are often missing or unstable: IDB (s) accumulates at runtime in varying delta sizes across iterations, and real-world applications such as DOOP [9] and DDISASM [15] often involve highly complex join

<sup>3</sup><https://materialize.com/blog/generalizing-linear-operators/#fusing-logic>

topologies (e.g. cyclic joins) than those typically handled by traditional DBMS. Hence systems Soufflé and DDlog give up and use only hard-coded listing orders: if a rule is written with body  $R, S, T$ , the system executes joins in that order even if there are cross products, i.e.  $(R \bowtie S) \bowtie T$ . RecStep collects runtime statistics periodically and invokes the DBMS’s optimizer to re-optimize query plans on the fly, paying a synchronization and catalog maintenance overhead.

FlowLog’s optimizer chooses a static join–project plan per rule and maps it one-to-one to the IR. Rather than relying on any runtime statistics, it analyzes the rule’s join graph using conventional heuristics (e.g., filter pushdown) plus worst-case-aware analysis [66]. Although not always optimal, this approach reliably avoids disastrous orders in our experiments (Sec. 10) and provides a principled base for future EDB-aware cost models.

## 5.1 Structural Cost Model

FlowLog adopts a structural cost model that uses the distinct number of participating variables as a proxy for the asymptotic cost of a transformation. For example, in Figure 2b, the lower Join-FlatMap involves  $x, y, z$  variables and hence bears a costing of 3, while the Join-FlatMap above involves  $z, x$  and is assigned a cost of 2.

We define the cost of a join-project plan (or equivalently, an IR) as the maximum cost of any transformation it contains. In Figure 2b, the maximum is 3, determined by the lower Join-FlatMap. Intuitively, for Example 2.1 over an input graph of  $n$  nodes, this plan implies a worst-case intermediate size (and so is the asymptotic runtime) of  $O(n^3)$ . The same reasoning extends to arbitrary Datalog programs: the structural cost upper-bounds worst-case intermediate output sizes, and formal guarantees are in [66]. Similar techniques are proposed for subgraph pattern matching [39] and have shown effectiveness for multi-way many-to-many joins.

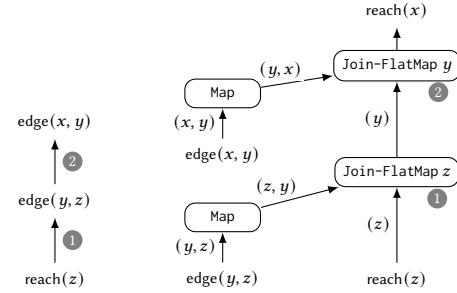
## 5.2 Search Strategy

We establish necessary definitions to describe the search space of join-project plans. The optimizer picks the plan in the search space that minimizes the cost under the structural cost model in Sec. 5.1.

A **(weighted) join graph** of a rule is a graph where every node is an atom and an edge exists if the two atoms join on at least one variable, and its weight is the number of variables they join on.

A **join tree** [63] is a spanning tree of the join graph such that for every variable  $x$ , the atoms containing  $x$  (in its schema) induce a connected subtree of the spanning tree. The join tree is then used to define acyclicity of a rule, i.e. a rule is acyclic if and only if there is a join tree for it. Zhao et al. [66] showed that for acyclic rules, bottom-up join orders (rooted arbitrarily) along a join tree, after early projections, usually yield tight intermediate size bounds.

A **join spanning tree** (JST) extends this notion to cyclic rules. A JST is a maximum spanning tree of the weighted join graph and reduces to a join tree when the rule is acyclic [37]. A **rooted JST** defines a join-project plan by post-order traversal: at each step, an atom is joined with its parent, followed by projecting out variables no longer needed. The only JST (also a join graph or a join tree) for Example 2.1 is shown in Figure 3. Here we root the JST at  $\text{edge}(x, y)$ , and it maps into the IR on the right, i.e. a bottom-up join order. If we root the JST at  $\text{reach}(z)$  instead, we recover the IR in Figure 2b.



**Figure 3:** A rooted JST for  $r_2$  in Example 2.1 (left) and its translated IR (right) following a post order traversal of the rooted JST.

**Search Space.** Our search space excludes semijoins (atoms whose variables are subsumed by another), antijoins and filters, as they are pushed down to the lowest possible transformation in the IR. This narrows down to a multi-way (inner) join where the search space is defined as *all rooted JSTs of its join graph*. There are three reasons for such a choice: (1) JSTs avoid cross products when possible, since cross products correspond to zero-weight edges in the weighted join graph, (2) this space is reasonably small and easy to enumerate [59], and (3) it collapses down to rooted join trees for acyclic rules, for which the post-order join-project plans yield tight bounds on the intermediate sizes (proven by [66]). In our running example (which is acyclic), Figure 3 will be selected against Figure 2b as it has a cost of 2, instead of 3 in our cost model. Intuitively, it avoids computing  $\text{edge}(x, y) \bowtie \text{edge}(y, z)$  by using two semi-joins.

Next, we show a more involved example and show that JSTs can optimize join-project plans for rules with cyclic multi-way joins.

*Example 5.1.* Consider an expensive rule (Figure 4, left) from the DOOP program analysis framework [9], which contains a recursive 8-way join. `VarType`, `HeapAllocationType`, and `ComponentType` are EDBs; others (in bold) are IDBs. Figure 4(center) shows its join graph. The JST selected by the optimizer is rooted at `LoadArrayIdx` and is annotated on the join graph by thick, directed edges. The dotted edges are edges in the join graph, but are not part of the rooted JST. Here, `Reach(inm)` is a semijoin atom to be subsumed by `LoadArrayIdx` and `inm` is projected away after semijoin. When creating the IR, this semijoin is pushed down to the leaf `LoadArrayIdx`.

We now discuss query plans for the rule in Example 5.1. Soufflé uses the given listing order: it joins `Reach` with `LoadArrayIdx`, then the result joins with `VarPointsTo`, and so on. The listing order here has been hand-picked for practical efficiency. In our cost model, the costing of this listing order is 5, dominated by the fifth join with `HeapAllocationType`: when we finish the first four joins up to `VarType`, the resulting schema, projecting away unnecessary variables, is  $(\text{bh}, \text{tp}, \text{heap}, \text{to})$ , where `bh` and `tp` are join keys for future joins with `HeapAllocationType` and `SupertypeOf` respectively;  $(\text{heap}, \text{to})$  are desired output variables. The next join with `HeapAllocationType` involves 5 variables, that is, `bh`, `tp`, `heap`, `to`, and `bht`. In contrast, the rooted JST in Figure 4(left) turns into the IR (Figure 4(right)),  $Jn$  being a shorthand for `Join-FlatMap`. The leaf `LoadArrayIdx \bowtie Reach` is the semijoin pushdown. The optimizer

favors this join-project plan as it has a cost of 3—no transformation needs more than 3 distinct variables—lower than the listing order.

### 5.3 Plan Execution

**Left-deep Plans.** Join planning is tied closely to the underlying execution. The listing order Soufflé, DDlog and others used is equivalent to a left-deep join plan in a DBMS and is incapable of representing bushy plans, such as the one in Figure 4. This is a necessary choice for Soufflé as it always compiles the listing order into an indexed nested loop join, where indexes (e.g. B-trees) are built on every atom except the first one and it iterates over each tuple of the first atom (as the outer loop) and probes into the indexes of the rest. Left-deep plans allow pipelining without intermediate materialization, which makes Soufflé memory-efficient. However, scaling indexed nested loop joins to a multi-threaded setting is non-trivial (e.g. [31]) and modern Datalog systems such as Soufflé and Flan [1] only distribute the outermost for-loop among threads. As shown in our experiments, this level of parallelism is insufficient to saturate resources even for simple recursive queries like reachability.

**Bushy Plans.** FlowLog targets DD, a fundamentally different execution model. Its differential operators are inherently stateful, so intermediate materialization is unavoidable. A key advantage, however, is their asynchrony: changes propagate through the dataflow graph and operators react concurrently without waiting for earlier stages to finish. Hence, DD naturally exploits multicore parallelism, as computation is scheduled dynamically based on available updates rather than a pre-determined execution sequence. Consequently, FlowLog can take advantage of bushy plans (i.e. IR is bushy) without worrying about compromising parallelism. To control memory blow-ups, FlowLog leverages insights from database theory (e.g. tree decompositions [28, 29, 66], worst-case optimal joins [43, 58], etc.) to select plans that have smallest possible worst-case intermediate sizes. When multiple candidate plans have the same estimated cost, FlowLog heuristically prefers bushier (shallower) rooted JSTs.

## 6 MAKING DATALOG ROBUST

Datalog execution typically exhibits high sensitivity to data distribution, recursive control, join orders, etc. The volatility makes conventional optimization techniques for SQL less effective, or even inapplicable. When a suboptimal plan is chosen for a recursive rule, the iterative nature of Datalog may exacerbate the slowdown. In contrast, there has been a growing interest in techniques that makes SQL queries robust, such as sideways information passing (sip) [23, 68], predicate transfer [62], and diamond hardened joins [7]. These approaches advocate a more pessimistic stance, prioritizing resilience against worst-case scenarios to ensure stable performance. However, robustness techniques are largely absent in Datalog systems. Integrating such techniques into FlowLog is a first step toward our vision: making Datalog execution robust. In fact, for a simple  $R \bowtie S$ , DD’s `join` operator already yields robustness as it runs a symmetric hash-join that balances both inputs (i.e. no build-probe asymmetry) [40]. For multi-way joins, the optimizer of FlowLog uses a sip-style algorithm (discuss next) to stabilize iterative execution.

**sip-style Algorithms.** The key idea of sip-style algorithms is to pre-filter dangling tuples before the actual joins. Dangling tuples are tuples that do not participate in the final output. The Yannakakis

algorithm [63] is a classic example. It requires the join graph to be acyclic to be able to construct a join tree. Following the post-order traversal over the join tree, it applies two sequence of semijoins to pre-filter base tables: (1) a bottom-up pass that semijoins the child atoms with the parent, followed by (2) a top-down pass that uses the reduced parent atoms to further semijoin-reduce the children. The semijoins provably prune all dangling tuples and the subsequent joins exhibit robustness against bad join orders in practice [62, 68].

Many expensive Datalog rules, however, involve cyclic join graphs, where no join tree exists. Inspired by Yannakakis, we apply a similar two-pass semijoin reduction in FlowLog, but for arbitrary join graphs. Our approach is as follows. We pick any atom in the join graph to start a breadth-first search (BFS). As we visit an atom, we semijoin-reduce it using its already-visited neighboring atoms, on the join keys. We conclude the first pass when all atoms are visited. Then we traverse the join graph in the reverse order of the first pass with another round of semijoin reduction (i.e. the second pass).

*Example 6.1 (Galen).* Consider the following program from [35] that describes an inference task in medical ontologies:

```
r1. p(x,z) :- p(x,y), p(y,z).
r2. p(x,z) :- p(y,w), u(w,r,z), q(x,r,y).
r3. p(x,z) :- c(y,w,z), p(x,w), p(x,y).
r4. q(x,r,z) :- p(x,y), q(y,r,z).
r5. q(x,u,z) :- q(x,r,z), s(r,u).
r6. q(x,e,o) :- q(x,y,z), r(y,u,e), q(z,u,o).
```

where the atoms  $u$ ,  $c$ , and  $s$  are EDBs. The listing orders here are hand-picked for performance on the given dataset [35]. Among them,  $r_2$ ,  $r_3$ ,  $r_6$  dominate the runtime and they are highly susceptible to bad join orders. The plan optimization in Sec. 5 can effectively handle  $r_2$ —it avoids joining  $u$  and  $q$  upfront (due to a high cost of 6 in our cost model). However, the optimal join orders for  $r_3$  and  $r_6$  are obscure (note that  $p, q$  are IDBS in a mutual recursion). Take  $r_3$  for example, it has a triangular join graph and all join orders are indistinguishable under our cost model (i.e. cost of 4) and yet the chosen order is an order of magnitude faster than, say,  $c(y,w,z)$ ,  $p(x,y)$ ,  $p(x,w)$  (we will call it the bad listing order henceforth).

We next show how `sip` is applied to  $r_3$  (Example 6.1). We implement it via rule rewriting: we pick  $c$  as the start and visits the atoms in the bad listing order. Then the rewritten `sip` rules for  $r_3$  are the following (underscores are placeholders for unused variables):

```
// (1) first pass c(y,w,z) -> p(x,w) -> p(x,y)
p1(x,y) :- c(y,_,_), p(x,y).
p2(x,w) :- c(_,w,_), p1(x,_), p(x,w).

// (2) 2nd pass c(y,w,z) <- p1(x,w) <- p2(x,y)
p3(x,y) :- p1(x,y), p2(x,_).
c4(y,w,z) :- c(y,w,z), p2( _,w), p3( _,y).

// (3) reduced join equivalent to the original r3
r3'. p(x,z) :- c4(y,w,z), p3(x,y), p2(x,w).
```

The rewriting is equivalent to  $r_3$ , but it is more robust against poor join orders. Indeed, the bad listing order incurs a substantial output blow-up when joining  $c$  and  $p(x,y)$ , but the final output shrinks significantly at the last join with  $p(x,w)$ . The rewriting, without such runtime knowledge, passes the selective semijoins of  $p2(x,w)$  to the first two atoms and reduces them into  $p3, c4$  in the second pass. Even though  $r'_3$  uses the same join order as  $r_3$ ,  $c4 \bowtie p3$  is now drastically smaller. Similar techniques can be applied to  $r_6$ .

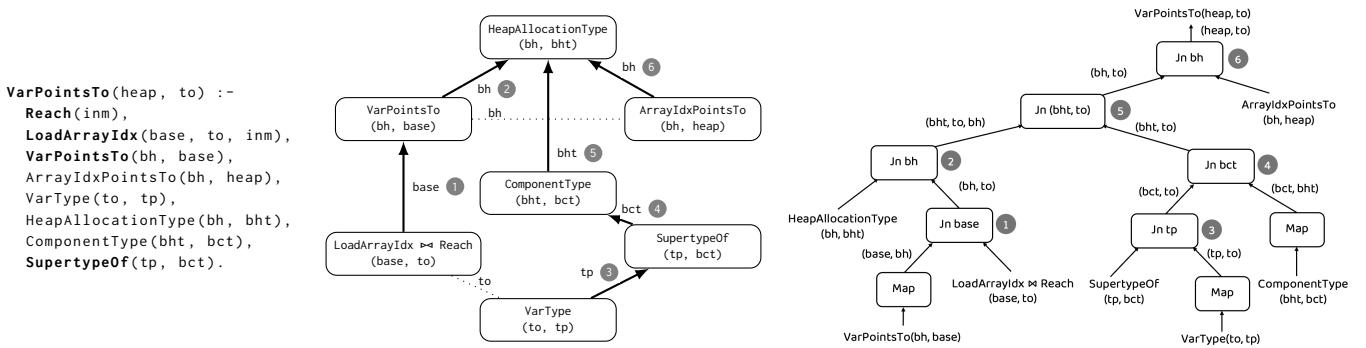


Figure 4: The doop rule for Example 5.1 (left); the rooted JST chosen by the optimizer over the cyclic join graph (center, numbered post-order); corresponding IR following the post-order (right). Semijoin `Reach(inm)` is pushed to `LoadArrayIdx`; `Jn` is a shorthand for `Join-FlatMap`.

**sip Overheads.** The sip rewriting introduces new intermediate (semijoin) rules, hence new IR (s): in this example, p1, p2, p3, c4, pre-filter the input tables for r3. This inevitably incurs overheads like maintaining these extra semijoin IR (s). However, this is outweighed if most of the dangling tuples are pruned. We will show in Sec. 10.4 that sip often leaves the structural optimizer (Sec. 5) with a near-worst-case scenario (i.e. all remaining tuples join), and by design, the optimizer strives for worst case optimalities. Future optimizations include incorporating techniques such as predicate transfer [62] to further mitigate the semijoin costs using lightweight Bloom filters.

## 7 SUBPLAN SHARING

McSherry et al. [40] introduced arrangements for DD to enable efficient index sharing across concurrent queries without redundant reconstruction. They demonstrated the benefits on simple Datalog programs by manually enforcing arrangement sharing.

In FlowLog, we extend this idea by automatically detecting and sharing common subplans both within and across rules to reduce memory consumption during execution. The sharing algorithm we use is greedy, exploiting the fact that the executor (and DD) will incrementally maintain the output of every intermediate operator. The input we have is a set of IR, one for each rule, where each IR is a logical plan with no sharing components yet. To identify reusable subplans, we normalize each IR in a canonical form and hash every subplan. If a duplicate hash is detected, the corresponding subtree is truncated and replaced by a pointer to the output of the first occurrence—a shared subplan possibly from the same or a different IR. We repeat this truncation until no more sharing can be found.

Figure 5 illustrates this process on the IR in Figure 3. The left simply canonicalizes this IR by encoding variable positions relative to their atoms. For example, in `edge(x, y)` and `edge(y, z)`, variables are rewritten as  $(e.0, e.1)$ , while `reach` maps to  $r.0$ . This reveals that two `Map` subplans are identical (up to variable renaming, having the same hash value, say  $0x42$ ). Thus, we reuse the output of the first `Map`, i.e. an index on key  $e.1$  and value  $e.0$ , for the second instance (as the leaf node). Figure 5(right) shows the resulting IR after sharing.

Our approach subsumes shared arrangements [40] and further extends it to sharing common table subexpressions (CTEs), e.g. if there is another IR requesting the first column  $e.0$  from `edge(x, y)`  $\bowtie$

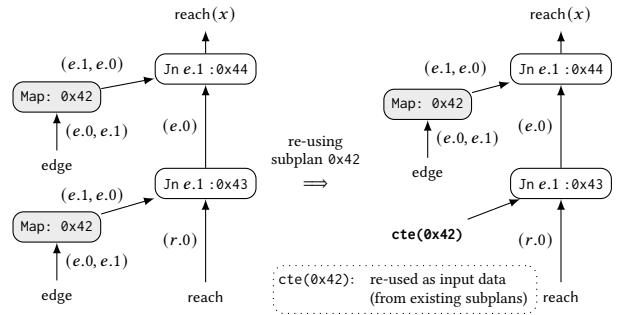


Figure 5: Subplan sharing within IR of Fig. 3 (reused Map are shaded)

`reach(y)`, the optimizer will not re-construct, but instead directly link to the output of the lower `Join-FlatMap` of Figure 5, say,  $0x43$ .

**Summary.** Figure 5(right) is the final IR for  $r_2$  (Example 2.1) that FlowLog’s optimizer sends to its executor. To summarize our optimizations (of each rule), we ① enumerate every rooted JST of the rule, chooses the best based on the cost model (Sec. 5), and maps it into an IR, ② create new IR (s) for auxiliary semijoins (or sip rules) to pre-filter atoms as reduced inputs for the original IR (Sec. 6). Along the way, we ③ aggressively fuse operators (Sec. 4), and ④ hash every subplan to maximize CTE reuse within and across IR.

## 8 BOOLEAN SPECIALIZATION

This section introduces a novel optimization of FlowLog’s executor. For each row (data, time, diff), DD uses an integral (e.g. 64 bits) `diff` to encode the number of copies of data, 0 being its absence and negative values indicating deletions or subtractions. Differential operators use integer arithmetics (e.g.  $+, -, \cdot$ ) to track output diffs. A `join` multiplies `diff`s of the two matching rows; `concat` of  $((a, b), 0, 4)$  and  $((a, b), 0, 3)$  results in  $((a, b), 0, 7)$  by adding the `diff`s, while `antijoin` yields  $((a, b), 0, 1)$  by subtracting the `diff`.

Integer arithmetic on `diff` fits well for incremental execution, but it is not always necessary for Datalog, where the mere presence of a tuple may be enough. In DD, it corresponds to restricting the `diff` to the Booleans, i.e. true for presence and false otherwise. Then, `join` implies a logical AND, as the output row is present only if both inputs are. A `concat` of  $((a, b), 0, \text{true})$  and

$((a, b), 0, \text{false})$  returns  $((a, b), 0, \text{true} \vee \text{false} = \text{true})$  by a logical OR—present when at least one exists. However, an `antijoin` is not expressible because subtraction is undefined for Booleans. In fact, DD encodes *presence* of a tuple as a zero-bit struct; a `false` (or *absent*) tuple is dropped upon encounter. A Boolean diff has two benefits: (1) it reduces memory footprint by storing diff as a zero-bit presence struct, and (2) the logical simplifications enable compiler optimizations that short-circuit Boolean operations.

As such, FlowLog enforces Boolean diffs (as the zero-bit presence struct) whenever possible by coercing the input diff values into Booleans and each differential operator, such as `join`, `concat`, into Boolean semantics, i.e. AND, OR operations. To handle non-monotonic operators (i.e. those that require negation or deletion operations), e.g. antijoins, we introduce a custom `lift` operator that transitions diff values between any data types, e.g. lift `true` to 1 and `false` to 0. In our example,  $((a, b), 0, \text{true})$  `antijoin`  $((a, b), 0, \text{true})$  is lifted to an integer subtraction  $((a, b), 0, 1 - 1 = 0)$ , and then casted down to a Boolean diff of `false`. As a general technique, FlowLog also supports diff field over other arithmetics (e.g. MIN) to express and optimize recursive aggregation (see next).

## 9 EXTENSIBILITY

**Algebraic Semantics.** A first extension (and optimization) is on the algebraic types for diff. Sec. 8 encodes diff as a Boolean value, to optimize batch Datalog queries. For *incremental Datalog* (EDBs have insertions/deletions over time), we fall back on integer arithmetic as DD normally does. Recursive aggregation arises frequently in modern data analytics [26, 36, 50, 57]. For example, this program computes connected components (CC) of an undirected graph:

```
cc(x, MIN(x)):- edge(x, _). // initialize labels i
cc(x, MIN(i)):- edge(y, x), cc(y, i). // propagate MIN(i)
```

The program iteratively propagates labels—discarding larger ones as smaller labels are found—until all nodes in a connected component share the same label. Semi-naïve execution can not handle this recursive MIN because prior facts may be retracted. In fact, most Datalog engines require invasive changes to support this (e.g. Soufflé lacks recursive aggregation to-date). In contrast, FlowLog needs only minor glue: it reuses DD’s native incremental machinery (cf. CC, SSSP in Table 1). Similar to the Boolean specialization (Sec. 8), FlowLog implemented this by baking aggregations directly into the diff field via a *monoid* [19]. Intuitively, it is a carrier set plus certain on-top operations; Booleans for batch Datalog were (`bool`,  $\vee$ ), antijoins or incremental Datalog use integers ( $\mathbb{Z}$ ,  $+/-\cdot$ ), while CC uses labels and a `MIN` operator ( $\mathbb{Z}$ , `MIN`). Selecting a monoid—and, when needed, applying `lift` to cast between monoids—yields efficient, general aggregation semantics without executor redesign.

**Distributed Execution.** A natural extension is scaling FlowLog to distributed environments. DD operators support scale-out execution by-design: they are sharded across workers (i.e. threads or machines) to attain resource saturation. Unlike Soufflé and RecStep, which are grounded in single-machine execution because of their designs, or BigDatalog [51] that carefully re-designs distributed execution for recursive settings, FlowLog, as future work, can seamlessly extend beyond single-node Datalog setups.

## 10 BENCHMARKING AND EXPERIMENTS

**Programs and Datasets.** We curate a benchmark suite that, to our knowledge, is the broadest yet for modern Datalog engines. It subsumes nearly all publicly available programs and datasets used in recent publications of RecStep, Soufflé, and DDlog [4, 14, 22, 34, 65], plus several new programs/datasets we created or harvested from popular open-source projects. The suite spans across graph analytics, business intelligence, and static program analysis; and stresses diverse recursion behaviors (multi-way joins, mutual, nonlinear, deep iterations, aggregations/antijoins, etc.): (1) Bipartite is a custom program that decides if a connected undirected graph is bipartite (i.e., two-colorable), starting from an initialized blue node:

```
red(y) :- edge(x, y), blue(x).
blue(y) :- edge(x, y), red(x).
answer() :- red(x), blue(x).
```

(2) Graph queries: single-source Reachability (Reach), Shortest Path (SSSP), Same Generation (SG), Transitive Closure (TC), and Connected Components (CC); (3) Program analysis: Andersen, CSPN, and CSDA (Context-sensitive Point-to and Dataflow Analysis) from RecStep’s suite [14] (so are the datasets); (3) Dyck represents Dyck-2 reachability [34] and uses the two largest CFPQ instances (kernel, postgre) [34]<sup>4</sup>; (4) Galen (Example 6.1 for medical ontologies), and CRDT (conflict-free replicated data types) come from McSherry’s blog<sup>5</sup>; (5) Polonius (alias analysis) for Rust’s borrow checker from an open-sourced project<sup>6</sup>; (6) DOOP [9], a popular Java analysis framework, featuring 136 rules with complicated recursions. Similar to Soufflé and Flan’s evaluation, the datasets are sampled among the largest from the DaCapo suite [8]; (7) DDISASM is a simplified disassembly analysis program from [15] and its datasets are synthesized from the widely used SMT solvers: CVC5 [6] and Z3 [12]. For all benchmarks, we use integer data type for inputs; string-valued attributes are pre-hashed to integers before Datalog execution.

**Competing Engines.** We compare against several state-of-the-art, open-source Datalog engines: (1) **Soufflé** (compiled)<sup>7</sup> with its mature optimizer [4, 22, 24, 54]; (2) **RecStep** [14], which attains stronger performance and scalability over similar designs such as BigDatalog; (3) **DDlog** [46], which shares the same DD backend as FlowLog but differs in design and lacks key optimizations (e.g. memory reduction) discussed in this paper, making it a fair baseline. We also report results from (4) **DuckDB** [45], and (5) **Umbra** [7]—two highly optimized, state-of-the-art database engines that can execute a subset of our Datalog workloads in SQL. DuckDB has very recently added USING KEY optimizations [5] for recursive CTEs, and Umbra implements worst-case optimal join [16] and employs query compilation which allows advanced loop compilations for recursive queries [53]. However, SQL has limited recursion syntax, and as [20] and Table 1 show, nearly half of our benchmarks cannot be directly executed on both databases due to unsupported mutual or nonlinear recursion. We thus exclude these cases from our evaluation.

Soufflé (compiled) and DDlog require per-program compilation, whereas others (e.g. FlowLog) are interpreters. Soufflé incurs a ~10s compilation for small programs and 30s for larger ones such as

<sup>4</sup>[https://formallanguageconstrainedpathquerying.github.io/CFPQ\\_Data](https://formallanguageconstrainedpathquerying.github.io/CFPQ_Data)

<sup>5</sup><https://github.com/frankmcsherry/dynamic-datalog>

<sup>6</sup><https://github.com/rust-lang/polonius>

<sup>7</sup>Soufflé interpreter [22] is in general 1.5× slower and hence excluded from the paper.

**Table 1:** Runtimes (seconds) for 4 and 64 threads (shown as 4|64). Lower is better. Per row, the best performance is colored in **blue** for 4 threads and **red** for 64 threads; TO = 900s timeout; OOM = out of memory. Unsupported cases (e.g., mutual or nonlinear recursion) state the reason in-cell.

Program, #rules	Dataset		FlowLog (4   64)	Souffle (4   64)	RecStep (4   64)	DDlog (4   64)	DuckDB (4   64)	Umbra (4   64)
CC, 2 [14]	livejournal		<b>46.0</b> <b>9.1</b>		90.0 28.1 196.1 116.2 121.2 96.8 88.94 26.5			
	orkut		67.9 <b>13.5</b>		122.5 26.6 307.7 185.7 46.2 29.0 <b>42.5</b> 23.3			
	arabic		<b>403.3</b> <b>67.7</b>	✖ recurs. aggregate	TO 252.0 TO TO 632.1 397.81 OOM OOM			
	twitter		TO <b>145.1</b>		TO 488.1 TO TO* TO 485.2 OOM OOM			
Reach, 2 [14]	livejournal		11.3 <b>5.1</b>	21.5 19.1 21.3 9.1 112.3 104.1 <b>6.7</b> 6.0 13.8 11.5				
	orkut		19.1 9.2	38.3 32.9 33.5 12.6 186.7 172.9 <b>11.0</b> <b>8.6</b> 21.9 18.8				
	arabic		87.7 40.8	206.8 179.7 264.9 63.7 TO TO <b>60.4</b> <b>35.0</b> 95.0 74.8				
	twitter		274.8 94.8	TO TO 543.5 101.7 TO TO <b>121.6</b> <b>62.9</b> 189.6 167.0				
SSSP, 2 [14]	livejournal		<b>13.5</b> <b>6.3</b>			205.2 152.2 144.0 75.5 17.5 14.8		
	orkut		<b>21.2</b> <b>9.1</b>			332.5 237.2 70.73 40.4 24.8 22.5		
	arabic		<b>99.8</b> <b>45.9</b>	✖ recurs. aggregate		TO TO TO TO 139.5 118.58		
	twitter		302.9 <b>105.1</b>			TO TO TO TO <b>250.0</b> 222.89		
TC, 2 [14]	G10K-0.001		78.2 8.5	112.7 39.7 127.0 69.5 209.2 106.4 75.2 78.5 <b>23.1</b> <b>6.1</b>				
	G20K-0.001		<b>542.8</b> <b>42.4</b>	629.6 249.2 703.7 282.7 TO TO 476.1 717.4 00M OOM				
	G40K-0.001		TO <b>305.7</b>	TO 668.5 TO TO TO TO TO OOM OOM				
SG, 2 [14]	G10K-0.001		<b>177.6</b> <b>18.6</b>	379.0 41.3 427.5 161.8 361.9 122.0 674.4 670.8 00M 00M				
	G20K-0.001		TO <b>90.7</b>	TO 404.5 TO TO TO TO TO 00M 00M				
	G40K-0.001		TO <b>815.9</b>	TO TO TO TO TO TO TO TO 00M 00M				
Bipartite, 4 [61]	netflix		27.8 9.3	107.6 103.8 34.8 <b>8.0</b> 174.0 153.2 <b>12.1</b> 10.62 19.3 16.83				
	roadca		<b>3.5</b> <b>1.3</b>	29.5 6.0 145.2 172.5 15.2 13.0 22.9 11.1 42.6 33.3				
	mag		306.7 <b>76.7</b>	393.9 306.2 489.4 103.6 TO TO <b>167.9</b> 175.5 192.7 171.5				
CSDA, 2 [14]	httpd		4.1 <b>1.3</b>	15.2 9.8 56.1 45.5 24.0 22.0 <b>3.9</b> 6.6 18.16 7.6				
	linux		<b>22.5</b> <b>6.4</b>	145.2 77.4 599.7 272.9 121.5 109.1 41.7 23.5 94.3 27.4				
	postgresql		<b>11.4</b> <b>4.9</b>	125.1 40.4 341.5 206.3 73.9 69.2 25.3 10.1 83.0 23.0				
CSPA, 2 [14]	httpd		112.6 <b>14.4</b>	<b>67.8</b> 50.9 382.4 154.3 319.3 290.3				
	linux		25.1 <b>4.8</b>	<b>20.9</b> 14.3 74.8 54.5 71.2 55.1		✖ mutual, nonlin.		
	postgresql		120.4 <b>15.0</b>	<b>76.7</b> 56.1 344.6 161.3 326.7 282.0			✖ mutual, nonlin.	
Andersen, 4 [14]	medium		<b>7.9</b> <b>4.2</b>	85.1 31.1 32.7 12.0 52.0 50.5				
	large		<b>16.0</b> <b>5.7</b>	187.8 69.6 63.7 17.4 107.4 103.0		✖ nonlinear recurs.		✖ nonlinear recurs.
Dyck, 7	kernel		<b>5.0</b> <b>1.4</b>	17.7 10.3 21.5 14.0 27.3 27.7				
	postgresql		<b>3.8</b> <b>0.9</b>	12.1 6.1 15.4 10.2 15.4 14.7		✖ nonlinear recurs.		✖ nonlinear recurs.
Galen, 8 [65]	galen		<b>32.2</b> <b>8.7</b>	59.3 36.8 486.5 667.9 111.6 64.6		✖ mutual, nonlin.		✖ mutual, nonlin.
CRDT, 23	crdt		248.3 <b>62.3</b>	177.7 230.2	✖ syntax error	TO 482.1 TO TO	<b>58.9</b> 132.8	
Polonius, 37	polonius		215.4 <b>41.4</b>	202.4 337.9	✖ syntax error	583.1 526.4 TO TO	<b>70.5</b> 67.1	
DDISASM, 28 [6, 12]	cvc5		87.5 <b>12.6</b>	<b>27.3</b> 14.5	✖ syntax error	438.6 111.3 TO TO		
	z3		<b>106.0</b> <b>27.9</b>	125.9 109.9		769.2 510.6 57.5 27.5		
DOOP, 136 [8]	batik		<b>65.2</b> <b>22.9</b>	651.1 160.2		151.4 126.6		
	biojava		<b>10.7</b> <b>7.7</b>	310.4 71.9		71.2 39.5		
	eclipse		<b>50.1</b> <b>18.2</b>	279.3 106.3	✖ syntax error	139.1 118.9	✖ mutual, nonlin.	✖ mutual, nonlin.
	xalan		<b>6.9</b> <b>6.3</b>	78.2 26.3		49.4 29.2		
	zxing		<b>10.5</b> <b>8.4</b>	89.7 31.9		57.5 27.5		

DOOP. DDlog exhibits much longer Rust compilation [22]—often >100s—even for small programs, due to heavy DD dependencies.

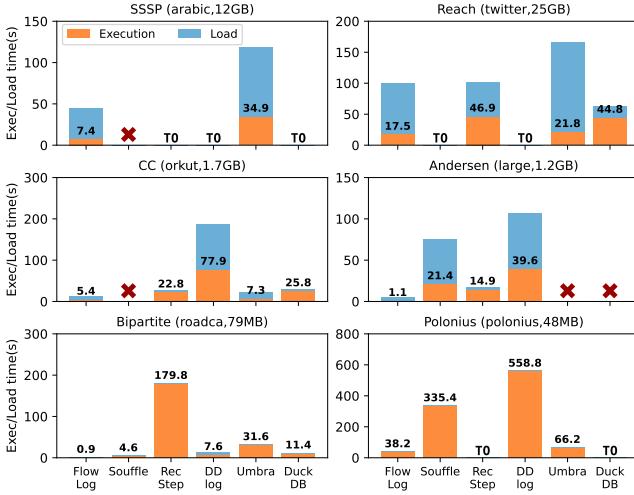
**Environment Setup.** We evaluate all engines in their latest releases on runtime, scalability, and memory usage. Experiments run on a CloudLab virtual machine [11] with two AMD EPYC 7543 32-core processors (64 physical cores, hyper-threading), running Ubuntu 22.04 LTS with 256 GB RAM.

## 10.1 Runtime Summary

Table 1 summarizes runtimes (in seconds) of all engines across our benchmark, using 4 and 64 threads. To account for join-order effects among Datalog engines, for each program-dataset pair we construct up to five plausible, semantically equivalent join-order variants (avoiding cross products); when fewer than five exist, we run all possibilities. We run every variant and report median runtimes for

each cell. For DuckDB and Umbra, we use semantically equivalent SQL; when multiple WITH RECURSIVE formulations are possible (e.g., USING KEY for CC in DuckDB), we take the best-performing one and report the median over five runs. The fastest per row is marked in **bold** (blue for 4 threads, red for 64 threads). As discussed in Sec. 5, each join-order variant is executed as a left-to-right binary-join plan in Soufflé and DDlog. RecStep reoptimizes join orders on the fly via its underlying DBMS optimizer; DuckDB/Umbra rely on their own optimizers for recursive CTEs. All FlowLog optimizations discussed in prior sections (Sec 4-9) are applied (including structural optimizations and sip), as a contrast to DDlog, being a baseline that directly translates Datalog into DD programs.

On 4 threads, FlowLog outperforms all competitors in 21 out of 41 program-dataset pairs. It consistently leads on programs such as Andersen, Dyck, and DOOP. For example, on Andersen (large),



**Figure 6: 64-thread runtime breakdown (s) for six program-dataset pairs.** Stacked bars show data loading (blue) and core Datalog evaluation (orange);  $\times$  are unsupported cases and T0 indicate 900s timeout. Numbers on top of each bar indicate core execution times only.

FlowLog runs  $11.7\times$  faster than Soufflé,  $4.0\times$  than RecStep, and  $6.7\times$  than DDlog. This advantage comes largely from logic fusion (Sec. 4) and subplan reuse (Sec. 7): necessary indexes for IDBs are built once and reused in multiple rules, avoiding a large amount of redundant work. That said, FlowLog does not always excel on **batch-oriented workloads**—programs having few but expensive iterations, such as Reach and CSPA. In such cases, its incrementality incurs overheads, as large intermediate results are maintained but used sparingly. Reach (twitter, 12 iterations) involves a single expensive join where DuckDB and Umbra’s highly optimized vectorized hash join implementations outperform FlowLog (Soufflé and DDlog timed out). Similarly, on CSPA (`htpd`, 29 iterations), Soufflé achieves the best performance at  $67.8s - 1.6\times$  faster than FlowLog’s  $112s$  runtime.

However, FlowLog exhibits markedly superior scalability compared to others. At 64 threads, it shows substantial speedups over all other engines and emerges as the fastest over 36 out of 41 cases! Even in previously disadvantageous workloads like CSPA, FlowLog demonstrates exceptional scaling (e.g.,  $7.8\times$  speedup on `htpd`), while Soufflé sees only modest gains (e.g.,  $1.3\times$ )—becoming  $3.5\times$  slower than FlowLog. This scalability advantage is consistently observed across workloads: on Polonius, Umbra’s runtime decreases marginally from  $70.5$  to  $67.1$  as thread count increases, FlowLog cuts down  $215s$  to  $41.4s$ . On DDISASM (`cvc5`), FlowLog transforms an initial  $3.2\times$  slowdown against Soufflé (at 4 threads) into a  $1.2\times$  speedup. Reach represents the sole exception where DuckDB maintains its lead; however, this advantage stems largely from optimized data loading rather than core execution performance (discuss next).

**Ablation Studies.** Table 1 reports end-to-end runtimes; Figure 6 breaks them into CSV loading (blue) and core Datalog evaluation (orange) for six workloads (large inputs: top two; medium: middle two; small: bottom two). FlowLog currently applies no ingestion-specific optimizations for CSV, so its loading time is generally higher than RecStep and DuckDB. In contrast, Soufflé and DDlog exhibit

long loading phases because they insert tuples one at a time into single-threaded indexed data structures (e.g., B-trees) before execution; for Andersen (1.2 GB input), their loading takes  $54.0s$  and  $66.7s$ , while FlowLog uses  $4.6s$ . Umbra can also be slow due to converting CSV files into its internal columnar format before execution.

FlowLog’s scaling-up performance is impressive: for all six benchmarks in Figure 6, its core execution is always the fastest. On Reach (twitter), although the end-to-end time trails DuckDB due to slower loading, its execution ( $17.5s$ ) is  $2.6\times$  faster than DuckDB’s  $44.8s$ . The smaller-but recursion-intensive bottom two workloads (i.e. execution time dominates) further underscore FlowLog’s core efficiency.

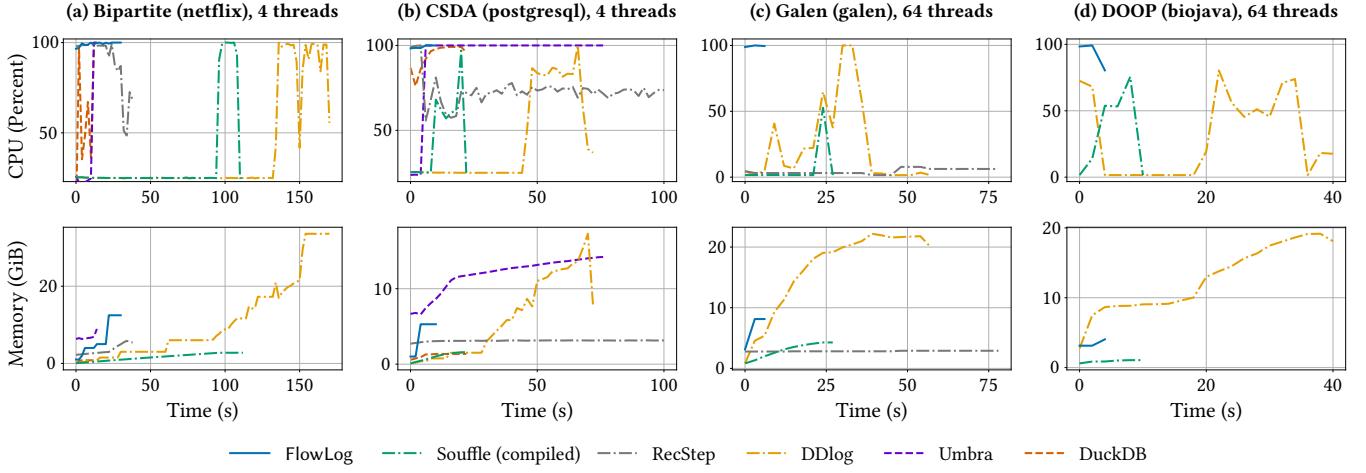
**Performance Analysis.** The consistently superior performance of FlowLog stems not from one, but from an ensemble of techniques in this paper. Logic fusion and subplan reuse are key mechanisms for controlling memory overhead in workloads generating explosive intermediate results (e.g., TC/SQ on G40K), where nearly all competing engines encounter timeouts or OOM (see Table 1). Structural query planning and `sip` collectively optimize (and stabilize) execution for recursive multiway joins, such as those in Galen and DOOP. Boolean/algebraic specializations (Sec. 8–9) make FlowLog competitive in recursive aggregates (in fact, achieving the best performance for both CC and SSSP in Table 1). FlowLog’s design inherits DD’s exceptional asynchrony, allowing it to excel in **long-tail workloads** (i.e. many lightweight iterations, typical in program analysis). The next sections provide detailed analysis for some of these findings.

## 10.2 CPU and Memory Usage

Figure 7 reports real-time CPU and memory usage of all systems on four workloads in Table 1 (FlowLog in blue solid curves).

**CPU.** For both 4 and 64 threads, FlowLog sustains near-100% CPU. Its high CPU usage stems from DD’s asynchrony, where all computation stages—including `iterate`—remain active and concurrent throughout execution (see Sec. 5.3), allowing a continuous resource saturation. DuckDB and Umbra, having mature parallelism infrastructures, also generally reach near-peak utilization. In contrast, Soufflé employs only outer-loop parallelism in its index nested loop joins; RecStep relies on the DBMS’s internal parallelism, but suffers from cross-iteration synchronizations. They both show moderate CPU usage for batch-oriented workloads such as Bipartite (Figure 7a, 4 iter.), but degrade sharply—often  $<25\%$ —on long-tail workloads such as Galen (Figure 7c, 32 iter.). Another inefficiency is evident in Figure 7a, where Soufflé and DDlog have long single-threaded phases for initial index construction. On large programs such as DOOP, which feature many rules of hybrid characteristics, Soufflé averages  $<50\%$  CPU usage. DDlog—lacking dedicated support for parallelism [13, 60]—displays volatile CPU consumptions.

**Memory.** A main inefficiency for DDlog is its large memory footprint, due to direct usage of DD without careful memory control. In all bottom panels of Figure 7, DDlog exhibits the highest memory usage ( $>20GB$ ). Umbra also shows elevated usage, especially in long-tail analyses (e.g., Figure 7b, 720 iter.) and triggers multiple OOM failures in Table 1. This is likely because Umbra does not have DuckDB’s `USING KEY` optimization [5] that inserts new facts in-place, and instead accumulates all results in memory. In contrast, DuckDB, Soufflé and RecStep incur lower memory overheads,



**Figure 7:** Live CPU (in %) and memory (in GBs) consumption on 4 different workloads; each panel shows CPU (top) and memory usage (bottom) over its execution horizon. Missing lines indicate unsupported cases (i.e. Umbra/DuckDB on Galen; Umbra/DuckDB/RecStep on DOOP).

generally staying <5GB. Although FlowLog builds on DD, our optimizations (Sec. 4–8) substantially shrink retained states: it uses average 3.5× less memory than DDlog on these workloads (e.g. 4 v.s. 20 GB on DOOP!), yet still 2–3× more than Soufflé’s lean baseline.

### 10.3 Parallel Scalability

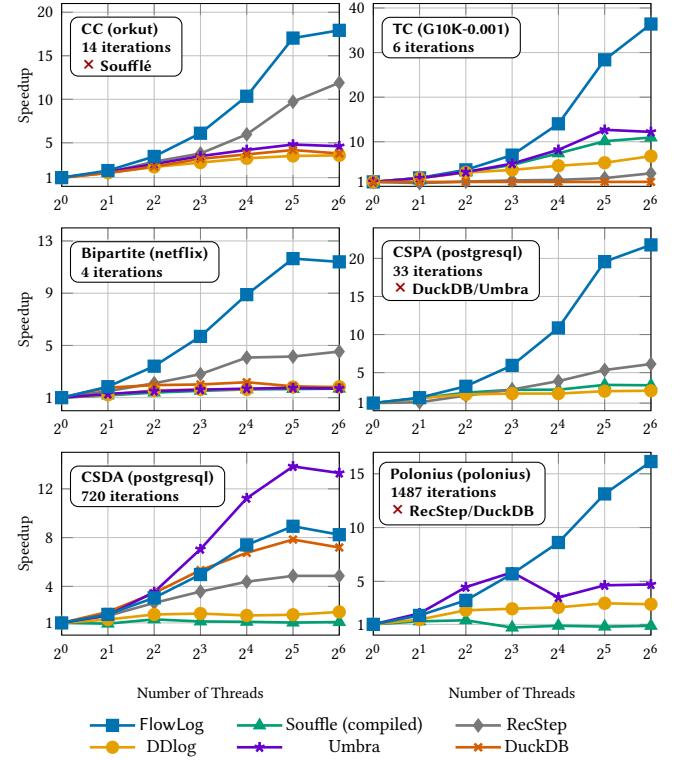
Scalability of Datalog engines is strongly workload dependent. Figure 8 presents the parallel speedups (over single-thread execution) of all systems across six representative cases: three graph queries (top row), and three program analyses (bottom row). FlowLog exhibits the most consistent scaling in all workloads: its speedup rises steadily through 32 threads and only mildly tapers at 64. This superior outscaling over the other engines (including DuckDB and Umbra) makes FlowLog the most competitive at high thread counts: at 64 threads, TC is the only one in six where another engine (i.e. Umbra, 6.1s) narrowly outperforms FlowLog (8.5s, a 38× speedup over its own baseline); DuckDB has almost no scaling in this case.

On long-tail workloads, parallelism may degrade; and overheads of thread management and data exchange may outweigh the benefits. We see this in CSDA (postgresql, 720 iterations) and Polonius (1487 iterations): Soufflé and DDlog gain almost no parallel speedup. In CSDA, DuckDB and RecStep scale slightly weaker and Umbra outpaces FlowLog, but they all start from a much slower single-thread baseline than FlowLog. This is largely due to these databases’ lack of continuously maintained views, forcing them to re-compute them at every iteration [13]. Umbra’s occasionally competitive scaling (e.g. CSDA) is due to its morsel-driven parallelism [31], which allows it to adaptively balance work across threads.

### 10.4 Structural Planning and Robust Execution

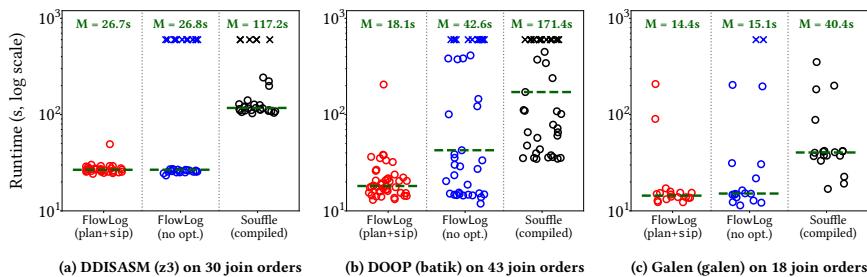
Next, we will study how join ordering impacts Datalog performance, and empirically validate our two complementary techniques: robust execution via semijoin prefiltering (or *sip* for short, see Sec. 6) and structural (worst-case) query planning (see Sec. 5).

First, we revisit RecStep’s reliance on its DBMS optimizer to optimize join orders on the fly. In programs such as Reach and Andersen,



**Figure 8:** Scalability of all competing systems on six program–data pairs; each subplot shows speedups relative to the single-thread run, up to  $2^6$  threads. Legends give the program–data pair and iterations to converge. Unsupported and timed-out cases are marked by  $\times$ .

where the optimal plan is either obvious (e.g., a single join) or plan choices have little performance variance, RecStep repeatedly invokes the optimizer at each iteration only to regenerate the same



**Figure 9: Runtime variability (64 threads) taking different listing orders.** Green numbers and dotted lines show the medians for FlowLog using the join optimizer and `sip`, FlowLog disabling both optimizations, and Soufflé. `T0` (i.e. >900s) and `OOM` cases are marked  $\times$ .

plan. This overhead erodes RecStep’s overall performance (Table 1). Galen further exposes its shortcomings: as data skew shifts across iterations, RecStep does not pivot promptly to a better order, spends many iterations in slow plans, causing eventual time-out.

Now, we demonstrate that FlowLog’s techniques in Sec. 5 and 6 jointly make it performant and robust. Figure 9 and Table 2 report runtime and plan sensitivity for three benchmarks having recursive multi-way joins: Galen, DOOP (batik), and DDISASM (z3).

**Performance and Robustness (Figure 9).** For each benchmark, we repeatedly sample rules (having recursive multiway joins) and rewrite their listing orders into new, unused variants. We exclude variants that introduce cross products, since Soufflé executes them verbatim and typically triggers time-outs. Figure 9 compares runtime across: (i) default FlowLog (having techniques from Sec. 5–6), (ii) FlowLog turning off both techniques, and (iii) Soufflé. To distinguish, we will refer to them as FlowLog (plan+`sip`), FlowLog (no opt.) and Soufflé for the rest of this section. We omit DDlog, as FlowLog (no opt.) can be regarded as a memory-optimized variant of it. DuckDB and Umbra lack support for these programs.

Figure 9 shows that both FlowLog (no opt.) and Soufflé are highly sensitive to join ordering. Across 91 distinct listing orders, FlowLog (no opt.) times out or `OOM` on 25 instances, and Soufflé times out on 23—over 25% of all cases! In FlowLog (no opt.), bad orders causes DD to maintain large incremental join results, leading to timeouts/`OOM`. Soufflé, while avoids materialization via pipelined execution (see Sec. 5.3), still suffers from long runtimes due to expensive nested-loop joins. In contrast, FlowLog (plan+`sip`) never times out or runs `OOM` on any plan, and its runtime distributions are much more stable.

The figure also reports median runtimes over all evaluated join orders (`T0/OOM` conservatively set to 900s, favoring FlowLog (no opt.) and Soufflé). In every benchmark, FlowLog (plan+`sip`) achieves the lowest median runtime. The largest gains appear in DOOP: 2.4× over FlowLog (no opt.) and 9.5× over Soufflé. These speedups stem from expensive multi-way joins where the structural planner (Sec. 5) selects substantially better—though not always optimal—plans than fixed listing orders. For example, for the rule of Example 5.1, poorly assigned orders cause both FlowLog (no opt.) and Soufflé to time out, whereas FlowLog (plan+`sip`)’s optimizer chooses the bushy plan in Figure 4 (right), completing the entire program in 11s.

Prog. rule	plan+ <code>sip</code> *	plan only	<code>sip</code> only	no opt.
z3, $r_{18}$	22.6s	19.3s	22.2s	21.5s
	20.5s	19.7s	22.0s	20.5s
	19.9s	20.8s	<del>300M</del>	<del>300M</del>
batik, $r_{118}$	14.6s	12.4s	15.1s	11.7s
	21.1s	<del>343.6s</del>	20.5s	<del>346.5s</del>
	21.4s	<del>343.7s</del>	18.7s	16.7s
gallen, $r_2$	8.8s	7.4s	8.5s	7.4s
	8.0s	6.8s	<del>300M</del>	<del>300M</del>
	10.2s	11.2s	<del>300M</del>	<del>300M</del>

**Table 2: Runtime variability (64 threads) taking every rule ordering for  $r_{18}$  in DDISASM,  $r_{118}$  in DOOP (batik), and  $r_5$  in Galen, comparing FlowLog (plan+`sip`), and FlowLog disabling the query planner, `sip`, or both.**

**Planning & `sip` are complementary (Table 2).** Structural planning often steers away from pathological join orders, but for complex multi-way joins where the optimal order remains elusive, `sip` prefilters dangling tuples and cushions poor structural choices. For example, taking the bad listing order of  $r_3$  in Example 6.1, FlowLog (no opt.) takes 186s to run and peaks at 144 GB; FlowLog (plan+`sip`), even though the structural optimizer cannot find a better order, uses `sip` pruning to finish in 86s with a 47 GB peak.

Table 2 further dissects this finding, showing that both planning and `sip` are necessary for FlowLog to be fast and robust. For each benchmark in Figure 9, we pick three representative recursive rules ( $r_{18}$ ,  $r_{118}$ ,  $r_2$ ) whose bodies are triangular (three-way) recursive joins; for each such rule we enumerate all three distinct binary join orders. The table reports FlowLog’s runtimes under four settings: full system, planner only, `sip` only, and both optimizations disabled. Skull face entries highlight massive slowdowns or memory blow-ups. We observe that: (i) the structural planner alone sometimes lands on pathological orders that inflates intermediates; (ii) `sip` alone adds little benefit on non-selective joins and makes no attempt to improve the join order (note that it also incurs moderate semijoin overheads, albeit by a small margin); and (iii) using both lets `sip` prefILTER inputs so the remaining joins behave close to a worst-case data distribution (well handled by the worst-case-oriented planner), yielding the overall fastest and most robust column in Table 2.

## 11 CONCLUSION

We developed FlowLog, a new Datalog engine that decouples the logical optimizations from the physical execution on DD’s operators. FlowLog’s IR-level planning and semijoin prefILTERing cut memory, stabilize recursion, and deliver up to order-of-magnitude speedups over DDlog, Soufflé, RecStep, DDlog, DuckDB, and Umbra across diverse benchmarks. Looking ahead, we plan to add robust cardinality estimation, cost-aware optimizers, compile-time optimizations, richer incremental Datalog features, and elastic scale-out execution.

## ACKNOWLEDGMENTS

We gratefully acknowledge Sam Arch, Frank McSherry, Kristopher Micinski, Thomas Neumann, and Yihao Sun for their stimulating discussions and helpful insights, which greatly shaped and inspired the design of FlowLog.

## 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.
- [2] Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. 2005. Diagnosis of asynchronous discrete event systems: datalog to the rescue! (*PODS '05*). Association for Computing Machinery, New York, NY, USA, 358–367.
- [3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.
- [4] 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.
- [5] Björn Bamberg, Denis Hirn, and Torsten Grust. 2025. How DuckDB is USING KEY to Unlock Recursive Query Performance. In *Companion of the 2025 International Conference on Management of Data (SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 31–34.
- [6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvcs: A Versatile and Industrial-Strength SMT Solver. In *TACAS (1) (Lecture Notes in Computer Science)*, Vol. 13243. Springer, 415–442.
- [7] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (2024), 3215–3228.
- [8] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zisman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM. <https://doi.org/10.1145/3669940.3707217>
- [9] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*. ACM, 243–262.
- [10] David C.Y. Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky E. Katahanas, Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, and Heidi Howard. 2024. Optimizing Distributed Protocols with Query Rewrites. *Proc. ACM Manag. Data* 2, 1, Article 2 (March 2024), 25 pages.
- [11] CloudLab 2018. <https://www.cloudlab.us/>.
- [12] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340.
- [13] Zhiwei Fan, Sunil Mallireddy, and Paraschos Koutris. 2022. Towards Better Understanding of the Performance and Design of Datalog Systems. In *Datalog (CEUR Workshop Proceedings)*, Vol. 3203. CEUR-WS.org, 166–180.
- [14] Zhiwei Fan, Jianqiao Zhu, Zuyi 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.
- [15] Antonio Flores-Montoya and Eric M. Schulte. 2020. Datalog Disassembly. In *USENIX Security Symposium*. USENIX Association, 1075–1092.
- [16] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 1891–1904.
- [17] Thomas Gilray, Arash Sahebolamri, Yihao Sun, Sowmith Kunapaneni, Sidharth Kumar, and Kristopher Micinski. 2024. Datalog with First-Class Facts. *Proc. VLDB Endow.* 18, 3 (Nov. 2024), 651–665. <https://doi.org/10.14778/3712221.3712232>
- [18] Todd J. Green, Molham Aref, and Grigorios Karvounarakis. 2012. LogicBlox, Platform and Language: A Tutorial. In *Datalog (Lecture Notes in Computer Science)*, Vol. 7494. Springer, 1–8.
- [19] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*. ACM, 31–40.
- [20] Anna Herlihy, Anastasis Ailamaki, Martin Odersky, and Amir Shaikhha. 2025. Language-Integrated Recursive Queries. *CoRR* abs/2504.02443 (2025).
- [21] Anna Herlihy, Guillaume Martres, Anastasia Ailamaki, and Martin Odersky. 2024. Adaptive Recursive Query Optimization. In *ICDE*. IEEE, 368–381.
- [22] Xiaowen Hu, David Zhao, Herbert Jordan, and Bernhard Scholz. 2021. An efficient interpreter for Datalog by de-specializing relations. In *PLDI*. ACM, 681–695.
- [23] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *ICDE*. IEEE Computer Society, 774–783.
- [24] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. Brie: A Specialized Trie for Concurrent Datalog. In *PMAM@PPoPP*. ACM, 31–40.
- [25] Bas Ketsman and Paraschos Koutris. 2022. Modern Datalog Engines. *Found. Trends Databases* 12, 1 (2022), 1–68.
- [26] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Convergence of Datalog over (Pre-) Semirings. In *PODS*. ACM, 105–117.
- [27] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2023. Convergence of Datalog over (Pre-) Semirings. *SIGMOD Rec.* 52, 1 (2023), 75–82.
- [28] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *PODS*. ACM, 13–28.
- [29] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another? In *PODS*. ACM, 429–444.
- [30] Monica S. Lam, Stephen Guo, and Jiwon Seo. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 278–289.
- [31] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morse-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.
- [32] Viktor Leis, Andrej Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.
- [33] Viktor Leis, Andrej Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.
- [34] Yuanbo Li, Kris Satya, and Qirun Zhang. 2022. Efficient algorithms for dynamic bidirected Dyck-reachability. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29.
- [35] John Liagouris and Manolis Terrovitis. 2014. Efficient Identification of Implicit Facts in Incomplete OWL2-EL Knowledge Bases. *Proc. VLDB Endow.* 7, 14 (2014), 1993–2004.
- [36] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *PLDI*. ACM, 194–208.
- [37] David Maier. 1983. *The Theory of Relational Databases*. Computer Science Press.
- [38] Muhammad Numair Mansur, Valentin Wüstholtz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *ISSTA*. ACM, 236–247.
- [39] Benjamin J. McMahan, Guoqiang Pan, Patrick Porter, and Moshe Y. Vardi. 2004. Projection Pushing Revisited. In *EDBT (Lecture Notes in Computer Science)*, Vol. 2992. Springer, 441–458.
- [40] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. 2020. Shared Arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.* 13, 10 (2020), 1793–1806.
- [41] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*.
- [42] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: a timely dataflow system. In *SOSP*. ACM, 439–455.
- [43] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16.
- [44] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *Proc. VLDB Endow.* 11, 6 (2018), 663–676.
- [45] Mark Raasveldt and Hannes Mühliesen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12–15, 2020, Online Proceedings*. www.cidrdb.org. <http://cidrdb.org/cidr2020/papers/p23-raasveldt-cidr20.pdf>
- [46] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog (CEUR Workshop Proceedings)*, Vol. 2368. CEUR-WS.org, 56–67.
- [47] Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher K. Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1198–1223.
- [48] Arash Sahebolamri, Thomas Gilray, and Kristopher K. Micinski. 2022. Seamless deductive inference via macros. In *CC*. ACM, 77–88.
- [49] 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 (CC '16)*. Association for Computing Machinery, New York, NY, USA, 196–206. <https://doi.org/10.1145/2892208.2892226>
- [50] Amir Shaikhha, Dan Suciu, Maximilian Schleich, and Hung Q. Ngo. 2024. Optimizing Nested Recursive Queries. *Proc. ACM Manag. Data* 2, 1 (2024), 16:1–16:27.
- [51] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carla Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *SIGMOD Conference*. ACM, 1135–1149.
- [52] Ahmedur Rahman Shovon, Yihao Sun, Kristopher Micinski, Thomas Gilray, and Sidharth Kumar. 2025. Multi-Node Multi-GPU Datalog. In *Proceedings of the 39th ACM International Conference on Supercomputing*. 822–836.
- [53] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB Endow.* 15, 5 (2022), 1119–1131.
- [54] 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.
- [55] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Sidharth Kumar, and Kristopher K. Micinski. 2025. Optimizing Datalog for the GPU. In *ASPLOS* (1).

- ACM, 762–776.
- [56] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) (ASE ’16). Association for Computing Machinery, New York, NY, USA, 320–331.
- [57] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing Recursive Queries with Program Synthesis. In *SIGMOD Conference*. ACM, 79–93.
- [58] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2 (2023), 150:1–150:23.
- [59] Paweł Winter. 1986. An Algorithm for the Enumeration of Spanning Trees. *BIT* 26, 1 (1986), 44–62.
- [60] Jiacheng Wu, Jin Wang, and Carlo Zaniolo. 2022. Optimizing Parallel Recursive Datalog Evaluation on Multicore Machines. In *SIGMOD Conference*. ACM, 1433–1446.
- [61] Renchi Yang. 2022. Efficient and Effective Similarity Search over Bipartite Graphs. In *Proceedings of the ACM Web Conference 2022*. 308–318.
- [62] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *CIDR*. www.cidrdb.org.
- [63] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. IEEE Computer Society, 82–94.
- [64] 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.
- [65] David Zhao, Pavle Subotic, Mukund Raghethaman, and Bernhard Scholz. 2021. Towards Elastic Incrementalization for Datalog. In *PPDP*. ACM, 20:1–20:16.
- [66] Hangdong Zhao, Shaleen Deep, Paraschos Koutris, Sudeepa Roy, and Val Tannen. 2024. Evaluating Datalog over Semirings: A Grounding-based Approach. *Proc. ACM Manag. Data* 2, 2 (2024), 90.
- [67] Hangdong Zhao, Paraschos Koutris, and Shaleen Deep. 2024. Evaluating Datalog via Structure-Aware Rewriting. In *Datalog-2.0*. Vol. 3801. CEUR-WS.org, 48–53.
- [68] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust. *Proc. VLDB Endow.* 10, 8 (2017), 889–900.