*TPLP*: Page 1–17. © The Author(s), 2026. Published by Cambridge University Press 2026
doi:10.1017/xxxxx


# *Efficiency of Analysis of Transitive Relations using Query-Driven, Ground-and-Solve, and Fact-Driven Inference*

Yanhong A. Liu John Idogun Scott D. Stoller Yi Tong
*Stony Brook University*
{liu,jidogun,stoller,yittong}@cs.stonybrook.edu

**Abstract**

Logic rules allow analysis of complex relationships to be expressed easily, especially for transitive relations in critical applications. However, understanding and predicting the efficiency of different inference methods remain challenging, even for simplest rules given different kinds of input data.

This paper analyzes the efficiency of all three types of well-known inference methods—query-driven, ground-and-solve, and fact-driven—along with their respective optimizations, and compares with optimal complexities for the first time, for analyzing transitive graph relations. We also experiment with rule systems widely considered to have the best performance. We analyze all well-known rule variants and widely varying input graphs. The results include precisely calculated optimal time complexities; comparative analysis across different inference methods, rule variants, and graph types; confirmation with performance experiments; as well as discovery of a performance bug.

## 1 Introduction

With rapid advances of LLMs in AI, but without assurance in their speedy question answering, rigorous logic inference has become even more important for critical applications. However, understanding drastically different inference methods and their efficiency on different kinds of input data has remained a significant challenge.

Logic rules allow analysis of complex relationships to be expressed easily and precisely, especially with recursive rules for transitive relations in critical applications, e.g., understanding program structures, flows, and dependencies (Smaragdakis and Balatsouras 2015; Flores-Montoya and Schulte 2020); enforcing security policies (Li and Mitchell 2003; Hristova et al. 2007); probing networking systems including social networks (Loo et al. 2009; Seo et al. 2013); and analyzing knowledge graphs in general (Bellomarini et al. 2018; Hogan et al. 2021).

Powerful rule systems allow analysis expressed using rules to be executed automatically without low-level programming. There are not only Prolog systems since the 1970s (Sterling and Shapiro 1994) and answer-set programming (ASP) systems since the 1990s (Gebser et al. 2012), but also more restricted Datalog systems since the 1980s (Maier et al. 2018) that have been regularly (re)created since the 2000s (Ketsman et al. 2022) due to increasing need of analysis, especially for transitive relations.

How to best utilize the power and efficiency of automatic inference in rule systems? It is regularly observed that different inference methods and rule systems can have significantly different performance, besides requiring different ways to express the analysis:

- The same analysis written differently using rules, as well as the same size of input data having different shapes, so to speak, can have drastically different running times. Different inference methods can tolerate these differences very differently.
- For applications written using rule languages together with other languages, where input data and analysis results are large, efficiency of passing data and results can also make a large difference, and may dominate the running time.

It is desirable to better understand and predict the efficiency of different inference methods, for different variants of rules running on different kinds of input data of different sizes.

This paper presents a systematic, precise, and detailed comparative analysis of the efficiency of computing transitive relations. We consider all three types of well-known inference methods—query-driven, ground-and-solve, and fact-driven—along with their respective optimizations, across different recursive rules and different input graph types, and compare with optimal complexities for the first time.

We also confirm analyzed results with experiments using rule systems widely considered to have best performances: Prolog system XSB (Sagonas et al. 1994; Swift et al. 2022) for query-driven, ASP system clingo (Gebser et al. 2019; clingo 2026) for ground-and-solve, and Datalog system Soufflé (Jordan et al. 2016; Souffle 2026) for fact-driven.

We use the most well-known rules for computing the transitive closure of arbitrary graphs, but our method of calculating optimal complexities and our analyses for all three inference methods and rule variants are general.

- The rules for transitive closure are also known for solving graph reachability, the most fundamental problem in analyzing deep relationships, especially with unbounded depth, which makes transitive closure well known to be beyond first-order logic (Grädel 1991).
- The rules capture both key features of analyses of transitive relations—join and recursion. Rules with more joins and recursions, over more relations with more arguments, reduce to join and recursion as in the rules for transitive closure (Liu and Stoller 2009; Liu 2013).
- The rules have all three variants of recursion—left, right, and double—and their significant impact on efficiency has not been analyzed in prior comparisons except for showing two curves on one ad hoc graph type (Liu et al. 2022; 2023b).

We also examine a wide variety of input graph types, including all from an extensive previous study (Brass and Wenzel 2019b).

We present precisely calculated optimal time complexities for all combinations of rule variants and graph types; comparative analysis across different inference methods for all combinations; and confirmation with detailed experiments. In particular, we show how the analyzed complexities explain the sameness and differences in actual query times precisely. These results help predict the efficiency of different inference methods and best utilize different rule systems. We also discovered a performance bug in XSB.

There has been significant bodies of work on both precise complexities and performance measurements, as discussed in Section 6. Our contributions include four main aspects that are studied systematically for the first time.

- A comparative analysis across all three types of well-known inference methods, especially with their respective optimizations, and their impact on the complexity and efficiency of analysis of transitive relations.
- Precise optimal time complexity calculation for inference for different rule variants and input graph types, not ignoring rule variants and considering only data sizes.
- Detailed running time measurements for all of compiling and loading programs, reading input data, querying or grounding and solving, and writing query results.
- Confirmation of measured times against analyzed efficiency across all inference methods, rule variants, and graph types, helping best utilize different rule systems and possible improvements.

The rest of the paper is organized as follows. Section 2 compares different inference methods for logic rules and different rule systems and languages supporting the different inference methods. Section 3 explains the rule variants and graph types analyzed and evaluated. Section 4 presents precise optimal time complexity analysis as well as analysis for different inference and optimization methods. Section 5 describes experimental results with key findings and analysis. Section 6 discusses related work, and as conclusion, directions for future work.

## 2 Comparing inference methods and rule systems

We consider all three types of drastically different inference methods, used by mostly separate communities of logic rule systems:

1. query-driven inference, also known as goal-directed search or top-down evaluation, the core of Prolog systems and variants (Sterling and Shapiro 1994);
2. ground-and-solve inference, the core of ASP systems (Gebser et al. 2012);
3. fact-driven inference, also called bottom-up evaluation, the core of Datalog systems and variants (Maier et al. 2018).

We distill and contrast the fundamentals of these different methods, along with their powerful optimizations to overcome their root sources of inefficiency. Their impacts on the efficiency of analysis of transitive relations are described in Sections 4 and 5.

### *2.1 Inference methods with powerful optimizations*

Despite significant differences, all inference methods are aimed at providing the best efficiency. It is the optimized inference that is to be compared with optimal complexities.

**Query-driven inference.** Query-driven inference performs top-down evaluation starting from the given query, searching through rules recursively—querying, in left-to-right order, hypotheses of rules whose conclusion matches the current query—until supporting facts for all queries are found. This goal-directed search allows rules to be used for not only logic programming, but also relational programming as with database languages and recursive programming as with functional languages (Sterling and Shapiro 1994).

However, simple top-down evaluation may not terminate for recursive rules over cyclic relationships, and may take exponential time on acyclic relationships. This is because

the same queries may be repeated infinitely or exponentially many times. Top-down evaluation with tabling solves this fundamental problem by remembering the queries performed and reusing the query results (Tamaki and Sato 1986; Chen and Warren 1996; Swift and Warren 2012).

**Ground-and-solve inference.** Ground-and-solve inference finds answers to queries by using grounding followed by solving (Kaufmann et al. 2016). Grounding instantiates rules, replacing variables in rules with values that contain no variables, yielding a propositional program. Solving then finds satisfying answers to the propositional program under stable model semantics, by using try-and-backtrack combinatorial search.

However, naive grounding and solving can lead to combinatorial explosion and even nontermination. Optimized grounding tries to consider only possible combinations of values that can match given or derived facts, not all combinations (Kaufmann et al. 2016). Advanced search uses conflict-driven clause learning (CDCL) with back-jumping instead of backtracking, skipping more areas of unfruitful search (Marques-Silva and Sakallah 1999; Gebser et al. 2012).

**Fact-driven inference.** Fact-driven inference performs bottom-up evaluation starting from facts, repeatedly applying rules—inferring new facts from conclusions of rules whose hypotheses match existing facts—until no more new facts can be inferred. This is a least fixed-point computation, with a larger set of facts inferred in each iteration using facts from the previous iteration, until a fixed point is reached.

Naive bottom-up inference from all facts in each iteration can result in heavily repeated computation. Well-known semi-naive evaluation considers only new facts added in each iteration, yielding drastic efficiency improvement (Bancilhon 1986). However, it may still have redundant computations in each iteration. Optimal computation using incrementalization with minimum increment considers only one fact in each iteration and avoids all redundant computations, and furthermore gives precise complexity guarantees (Liu and Stoller 2009; Liu 2013).

### *2.2 Rule systems and languages*

For confirming analyzed inference efficiency through experiments, we use state-of-the-art rule systems widely considered to have the best performance using different inference methods: Prolog system XSB (Sagonas et al. 1994; Swift et al. 2022) for query-driven, ASP system clingo(Gebser et al. 2019; clingo 2026) for ground-and-solve, and Datalog system Soufflé (Jordan et al. 2016; Souffle 2026) for fact-driven.

**Unique advantages.** Beyond their shared power, rule systems implementing different inference methods have different unique advantages that are critical for different kinds of applications.

- Prolog systems support general-purpose programming. Also, their goal-directed search can answer queries (e.g., only paths from a particular vertex) much more efficiently by inferring only needed facts, not all facts.
- ASP systems allow powerful constraint programming and solving for constraint satisfaction and optimization problems (e.g., the traveling salesman problem).

| | Query-driven | Ground-and-solve | Fact-driven |
|---|---|---|---|
| inference method | top-down evaluation | propositions and satisfiability | bottom-up evaluation |
| optimized inference | tabling | optimized grounding and CDCL | semi-naive; minimum increment |
| rule systems | Prolog and variants | ASP | Datalog and variants |
| viewed as most efficient | XSB | clingo | Soufflé |
| unique advantages | general programming, query-restricted search | constraint programming and solving | generating standalone code from rules |

Table 1: Inference methods and rule systems.

- Datalog systems often generate standalone code specialized for the given rules (e.g., code for computing transitive closure) without requiring a general inference system at runtime, e.g., Soufflé does so, but XSB and clingo do not.

Table 1 summarizes the inference methods and rule systems. Note that other systems, e.g., Ciao and GNU Prolog, can also generate standalone code and support CLP for writing constraints, but they are less efficient; GNU Prolog even requires manual programming for tabling to avoid nontermination or exponential time for recursive rules on graphs.

**Rule languages.** Largely due to different inference methods used, rule languages supporting these methods have different features. We show this using the transitive closure problem.

Given a predicate, `edge`, that asserts whether there is an edge from a first vertex to a second vertex, the transitive closure problem defines a predicate, `path`, that asserts whether there is a path from a first vertex to a second vertex by following the edges. This can be expressed in all dominant logic languages, including Prolog, ASP, and Datalog variants, as follows:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Often, the name `reachable` is used in place of `path`.

Besides the two rules above, different rules languages require different additional specifications.

- **Prolog systems like XSB.** To avoid nontermination or exponential time, tabling directives are needed, as follows for using default tabling for predicate `path`:

  ```
  :- table path/2.
  ```

  Also, to return all facts of `path` with query-driven inference, additional rules like the following can be used, employing `fail` to continue the search for each next `path` fact until no more can be found and then the last `printPath` succeeds.

  ```
  printPath :- path(X,Y), write(path(X,Y)), fail.
  printPath.
  ```

- **ASP system clingo.** With ground-and-solve, by default all facts will be returned. To get back only facts of `path`, the following additional directive can be used:

```
#show path/2.
```

- **Datalog system Soufflé.** To compile Datalog rules to standalone code in C++, which is then compiled and run, additional type and input-output declarations are needed:

```
.decl edge(x:number, y:number)
.input edge
.decl path(x:number, y:number)
.output path
```

Additionally, Prolog systems like XSB are highly powerful and expressive, and can be used for direct scripting and timing for performance analysis. In contrast, ASP and Datalog languages like clingo and Soufflé have more limited features and rely on a host language—Python for clingo and C++ for Soufflé—for application development.

## 3 Rule variants and input graph types

We consider subtly different rule variants and a wide variety of graph types that may affect efficiency of inference methods, drastically.

**Variants of rules.** It is long known that different variants of rules for solving the same problem can have drastically different performance, but this has not been studied precisely in general. We consider all three well-known variants of rules for transitive closure:

1. Left recursion: the recursive rule can be written differently, by switching the two predicates, so that the recursive occurrence of `path` is on the left:

```
path(X,Y) :- path(X,Z), edge(Z,Y).
```

2. Right recursion: the recursive rule as written in Section 2 has recursive occurrence of `path` on the right.
3. Double recursion: the recursive rule can also be written by replacing `edge` with `path`, yielding two recursive occurrences of `path`:

```
path(X,Y) :- path(X,Z), path(Z,Y).
```

**Kinds of input graphs.** We consider and precisely define 12 different kinds of graphs, shown in Table 2. It includes all 11 kinds in rbench (Brass and Wenzel 2019b), created to address the issues with input in OpenRuleBench (Liang et al. 2009). We created the last one, grid, with more rigid and intertwined edges. Together, they aim to capture graphs of different shapes that may affect the efficiency of computing the transitive closure very differently.

Note that `n` is just one of the parameters, and is not always the number of vertices in the graph, although it is for the first 5 graph types in Table 3. Table 3 columns 2 and 3 show the precise numbers of vertices and edges.

| Name | Definition, fully and precisely captured by the set of edges | BW 2019 |
|---|---|---|
| `Cmpl`$_{\texttt{n}}$ | complete graph with vertices `1..n`,<br>with an edge from every vertex to every vertex<br>`{(i,j): i in 1..n, j in 1..n}` | $K_n$ |
| `MaxAcyc`$_{\texttt{n}}$ | maximum acyclic graph with vertices `1..n`,<br>with an edge from every vertex to every higher-indexed vertex<br>`{(i,j): i in 1..n-1, j in i+1..n}` | $T_n$ |
| `Cyc`$_{\texttt{n}}$ | cycle going around vertices `1..n` in order<br>`{(i,i+1): i in 1..n-1} + {(n,1)}` | $C_n$ |
| `CycExtra`$_{\texttt{n,k}}$ | `Cyc`$_{\texttt{n}}$ with `k` extra edges from each vertex `i` to `k` vertices so that these `k+1` vertices are evenly spaced on the cycle<br>`{(i,(i-1 + t*n/(k+1)) mod n + 1): i in 1..n, t in 1..k} + Cyc`$_{\texttt{n}}$ | $S_{n,k}$ |
| `Path`$_{\texttt{n}}$ | path going through vertices `1..n` in order<br>`{(i,i+1): i in 1..n-1}` | $P_n$ |
| `PathDisj`$_{\texttt{n,k}}$ | `k` disjoint paths, each going through `n` vertices in order<br>`{(i,i+k): i in 1..(n-1)*k}` | $M_{n,k}$ |
| `Grid`$_{\texttt{n}}$ | Grid of `n`-by-`n` vertices in order, with edges toward higher-indexed vertices<br>`{(j, j+1): i in 1..n, j in (i-1)*n+1..i*n-1} +`<br>`{(j, j+n): i in 1..n-1, j in (i-1)*n+1..i*n}` | – |
| `BinTree`$_{\texttt{h}}$ | complete binary tree of `h` levels of vertices with edges toward the leaves<br>`{(i,2*i): i in 1..2`$^{\texttt{h}-1}$`} + {(i,2*i+1): i in 1..2`$^{\texttt{h}-1}$`}` | $B_h$ |
| `BinTreeRev`$_{\texttt{h}}$ | complete binary tree of `h` levels of vertices with edges toward the root<br>`{(2*i,i): i in 1..2`$^{\texttt{h}-1}$`} + {(2*i+1,i): i in 1..2`$^{\texttt{h}-1}$`}` | $V_h$ |
| `X`$_{\texttt{n,k}}$ | X shaped with an edge from each of `n` vertices to a central vertex `n+1`, and an edge from the central vertex to each of `k` vertices<br>`{(i,n+1): i in 1..n} + {(n+1,n+1+j): j in 1..k}` | $X_{n,k}$ |
| `Y`$_{\texttt{n,k}}$ | Y shaped with an edge from each of `n` vertices to a central vertex `n+1`, and a path going through `k` vertices starting from the central vertex<br>`{(i,n+1): i in 1..n} + {(i,i+1): i in n+1..n+k-1}` | $Y_{n,k}$ |
| `W`$_{\texttt{n,k}}$ | W shaped subgraphs, formed with two sets of `n` vertices each, with an edge from each vertex in the first set to `k` vertices in the second set<br>`{(i,n+1 + (i+j-1) mod n): i in 1..n, j in 1..k}` | $W_{n,k}$ |

Table 2: Kinds of input graphs and their definitions, with their names if any in Brass and Wenzel (2019b).

## 4 Analysis of optimal time complexities

To compute all facts that can be inferred, any inference using bottom-up, group-and-solve, or top-down methods must consider all combinations of given and inferred facts that can make all hypotheses of a rule become true. Each such combination leads to a rule firing.

Optimality in considering such combinations can be achieved by using the method of incrementalization with minimum increment (Liu and Stoller 2009; Liu 2013). The method allows each combination to be considered at most once and in worst-case $O(1)$

| Graphs | #vertex | #edge | #path | Left recursion and right recursion | Double recursion |
|---|---|---|---|---|---|
| $\texttt{Cmpl}_{\texttt{n}}$ | $n$ | $n^2$ | $n^2$ | $n^3$ | same as left |
| $\texttt{MaxAcyc}_{\texttt{n}}$ | $n$ | $\frac{1}{2}n(n-1)$ | $\frac{1}{2}n(n-1)$ | $\frac{1}{6}n(n-1)(n-2)$ | same as left |
| $\texttt{Cyc}_{\texttt{n}}$ | $n$ | $n$ | $n^2$ | $n^2$ | $n^3$ |
| $\texttt{CycExtra}_{\texttt{n,k}}$ | $n$ | $n+nk$ | $n^2$ | $n^2+n^2k$ | $n^3$ |
| $\texttt{Path}_{\texttt{n}}$ | $n$ | $n-1$ | $\frac{1}{2}n(n-1)$ | $\frac{1}{2}(n-1)(n-2)$ | $\frac{1}{6}n(n-1)(n-2)$ |
| $\texttt{PathDisj}_{\texttt{n,k}}$ | $nk$ | $(n-1)k$ | $\frac{1}{2}n(n-1)k$ | $\frac{1}{2}(n-1)(n-2)k$ | $\frac{1}{6}n(n-1)(n-2)k$ |
| $\texttt{Grid}_{\texttt{n}}$ | $n^2$ | $2n(n-1)$ | $\frac{1}{4}n^2(n+3)(n-1)$ | $\frac{1}{2}n(n^3-5n+4)$ | $\frac{1}{36}(n^3+7n^2+2n-22)n^2(n-1)$ |
| $\texttt{BinTree}_{\texttt{h}}$ | $2^h-1$ | $2^h-2$ | $2^h(h-2)+2$ | $2^h(h-3)+4$ | $2^{h-1}(h^2-5h+8)-4$ |
| $\texttt{BinTreeRev}_{\texttt{n}}$ | $2^h-1$ | $2^h-2$ | $2^h(h-2)+2$ | $2^h(h-3)+4$ | $2^{h-1}(h^2-5h+8)-4$ |
| $\texttt{X}_{\texttt{n,k}}$ | $n+k+1$ | $n+k$ | $n+k+nk$ | $nk$ | same as left |
| $\texttt{Y}_{\texttt{n,k}}$ | $n+k$ | $n+k-1$ | $\frac{1}{2}(2n+k-1)k$ | $\frac{1}{2}(2n+k-2)(k-1)$ | $\frac{1}{6}(3n+k-2)k(k-1)$ |
| $\texttt{W}_{\texttt{n,k}}$ | $2n$ | $nk$ | $nk$ | 0 | same as left |

Table 3: Optimal number of combinations for all 3 recursion variants and all 12 graph types.

time—by considering only one fact at a time and using indexing to find each new combination with the fact in $O(1)$ time.

### *4.1 Optimal number of combinations*

For computing all `path` facts, for all three rule variants, the first rule is fired once for each `edge` fact, totaling to be `#edge`, the number of edges. For the second, recursive rule, we calculate the exact number of combinations for each graph type and each rule variant. The calculation, for each graph type and each rule variant, first forms a precise formula for the exact counting and summing, and then simplifies the formula to a closed form using Mathematica. Details of the calculation are in an appendix of Liu et al. (2026).

Table 3 shows the resulting precise complexity formulas in closed forms for the recursive rule. The particular closed forms in Table 3 were selected and rearranged manually to have a consistent factor representation across all rule variants and graph types.

Despite all the differences among the kinds of graphs listed, there are a number of interesting equalities to observe. For example, for `Cmpl` graphs, the number of combinations is the same for all different recursions, and this is also the case for `MaxAcyc`, `X`, and `W` graphs.

It is particularly interesting to see that for all graph types, left and right recursions lead to the same number of combinations. This is easy to see for graphs with a kind of

symmetry between paths going into an edge (left recursion) and paths coming out (right recursion), which include the first 7 graph types. It is also easy to see for `X` and `W` graphs, which are the simplest cases with only paths of lengths 2 and 1, respectively.

However, $\texttt{BinTree}_\texttt{h}$ and $\texttt{BinTreeRev}_\texttt{h}$ do not have such symmetry, similar to $\texttt{Y}_{\texttt{n,k}}$ not having such symmetry. It still happens to be that even for left or right recursion alone, $\texttt{BinTree}_\texttt{h}$ and $\texttt{BinTreeRev}_\texttt{h}$ have the same number of combinations despite they are from very different summation formulas. Consider the right recursion form `edge(X,Y), path(Y,Z)`.

- For $\texttt{BinTree}_\texttt{h}$, consider `i-1` levels of edges from root to leaves, with `i` from `1` to `h-1`. At the `i`th level of edges, there are $2^\texttt{i}$ edges. For each such edge `(u,v)`, there is a path from `v` to each vertex in the subtree rooted at `v` except for `v`. The subtree has $2^{\texttt{h}-\texttt{i}}\texttt{-1}$ vertices, and the connecting vertex itself should also be subtracted. Thus, the number of combinations for `edge(X,Y), path(Y,Z)` is $\sum_{i=1..h-1} 2^i * (2^{h-i} - 2)$.
- For $\texttt{BinTreeRev}_\texttt{h}$, consider `i-1` levels of edges from leaves to root, with `i` from `h` to `2`. At the `i`th level of edges, there are $2^{\texttt{i}-1}$ edges. For each such edge `(u,v)`, there is a path from `v` to each vertex above it in the path to root. There are `i-2` such vertices. Thus, the number of combinations for `edge(X,Y), path(Y,Z)` is $\sum_{i=h..2} 2^{i-1} * (i - 2)$.

Both actually simplify to the same closed form as in Table 3. Because of this same closed form, left and right recursions for both kinds of trees have the same closed form too. When we found that left and right recursions for `Y` graphs also have the same closed form, it became curious, and we prove the following general result for directed trees; the result actually applies to $\texttt{Path}_\texttt{n}$, $\texttt{PathDisj}_{\texttt{n,k}}$, and the last 5 graph types.

We prove that for any directed tree, left and right recursions have the same number of combinations. Precisely, let `vert` be the set of vertices in the given `edge` relation, then

$$\sum_{\texttt{v}\in\texttt{vert}} \#\{\texttt{u: path(u,v)}\} \times \#\{\texttt{w: edge(v,w)}\} = \sum_{\texttt{v}\in\texttt{vert}} \#\{\texttt{u: edge(u,v)}\} \times \#\{\texttt{w: path(v,w)}\}$$

For any directed tree, there is at most one path between any two vertices. Consider any matching `path(u,v)` and `edge(v,w)` from left recursion. Let `edge(u,v1)` be the unique edge on the path from `u` to `v`. Then there is the unique `path(v1,v)` joining with `edge(v,w)` giving the unique `path(v1,w)`. Now the matching `edge(u,v1)` and `path(v1,w)` must be counted in right recursion. Thus each combination `path(u,v)` and `edge(v,w)` from left recursion corresponds to a unique combination `edge(u,v1)` and `path(v,w)` from right recursion. Symmetrically, each combination from right recursion corresponds to a unique combination from left recursion. Thus, the equation holds.

Note that the equation does not hold for all graphs. An example is a graph with edges `{(1,2),(2,1),(1,3),(2,3)}`. The combinations `path(1,1),edge(1,2)` and `path(1,1),edge(1,3)` are not captured in right recursion, because there is no edge `(1,1)` or `(2,2)`.

### *4.2 Time complexities of different inference methods*

Despite that optimal complexities are achieved with incrementalization-based method (Liu and Stoller 2003; 2009) and confirmed experimentally (Liu and Stoller 2009), implementations using other inference methods have different other cost issues.

**Query-driven inference with tabling** has separate overhead from general mechanisms for table maintenance and lookup. However, this is minimal, but the following is a significantly larger cost issue.

The inference considers hypotheses of a rule from left to right, so the order of the two predicates matters. Contrary to common belief, left recursion is the most efficient for table lookup, as analyzed precisely in Tekle and Liu (2010); the key idea is that, with left recursion, the search for `path(X,Y)` looks up `path(X,Z)` in the same table due to the same value of `X`. With right recursion, it needs to look up `path(Z,Y)` in a different table for each different value of `Z` from `edge(X,Z)` on the left; this is also the case with double recursion except that the `Z` is from `path(X,Z)`, which could be asymptotically larger than from `edge(X,Z)` depending on the input graph.

**Ground-and-solve with optimized grounding and solving** generally has a separate cost tracking ground rules rather than directly returning the derived facts. This is not minimal but relatively small. For recursive rules with no negation, grounding does iterative evaluation on ground rules and computes all resulting facts.

A larger cost issue is similar to the issue with recursion variants for query-driven inference above, because grounding also considers hypotheses in a rule from left to right (Kaufmann et al. 2016). Additionally, despite extensive optimizations, iterative evaluation on ground rules uses semi-naive evaluation (Kaufmann et al. 2016), another larger cost issue as described for fact-driven inference below.

**Fact-driven inference with semi-naive evaluation** has redundant computations and overhead from considering all new facts in each iteration. This is because it goes through an extra set, and there could be repeated computations within the processing of a set instead of one element (Cai and Paige 1988), unlike incrementalization with minimum increment (Liu and Stoller 2009; Liu 2013). It could be asymptotically worse than optimal without prudent incremental processing of the extra set.

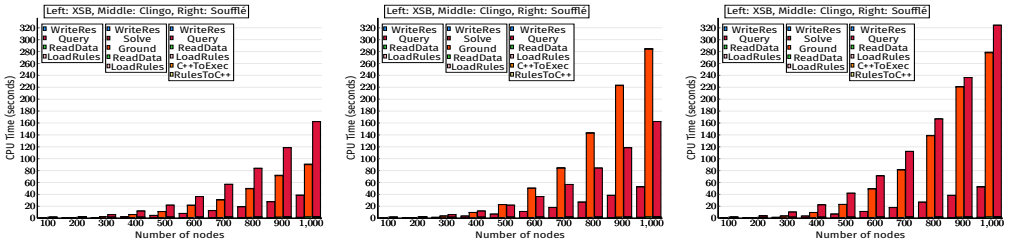
However, in contrast to the two methods above, inference driven by newly inferred facts of a predicate, e.g., `path`, can have the same efficiency regardless of whether the predicate is in a left or right recursion to join with another predicate.

Finally, fact-driven inference often generates standalone code from rules, which has a separate cost of compilation, but this is needed only once for each program.

Note that the precise formulas in Table 3 support comparisons of not only asymptotics but also constants. For example, for `Cmpl` graphs, all three recursion variants have exactly the same number; in comparison, for `MaxAcyc` with about 1/2 the size of input (`edge`) and output (`path`), left and right recursions have about 1/6 of the number of combinations, whereas double recursion has the same number. Section 5 shows through experiments how difference inference methods and systems match the analytical results in this section.

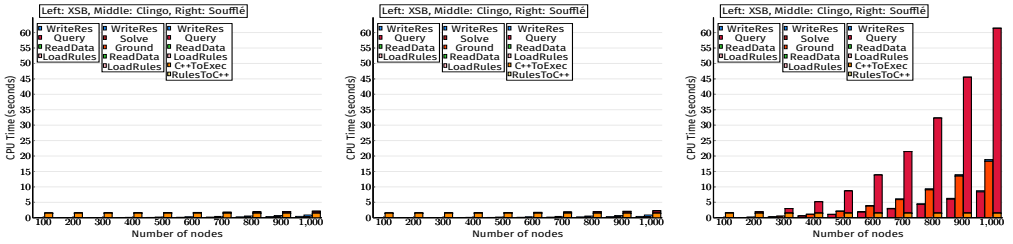
## 5 Key findings from experimental results and analysis

We designed and performed detailed experiments with all three rule systems XSB, clingo, and Soufflé, all three rule variants, and all 12 input graph types. The programs, variants of each, and graphs are exactly as described in Sections 2.2 and 3. For example, Figs. 1 and 2 show running times on `Cmpl` and `Path` graphs for all recursion types in all three rule



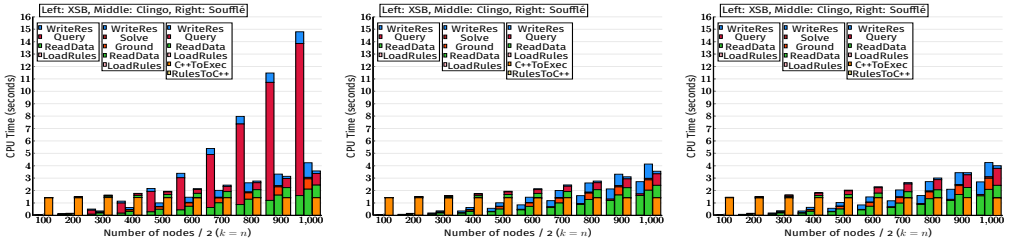

(a) Left recursion (b) Right recursion (c) Double recursion

Fig. 1: Running times on `Cmpl` graphs, up to 1000 vertices, 1000000 edges.




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. 2: Running times on `Path` graphs, up to 1000 vertices, 999 edges.

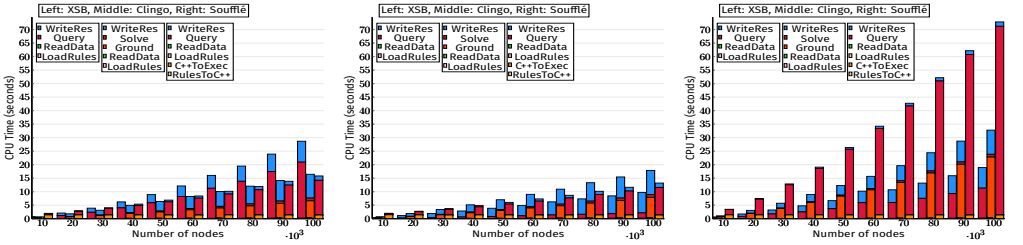



(a) Left recursion (b) Right recursion (c) Double recursion

Fig. 3: Running times on `W` graphs, up to 2000 vertices, 1000000 ($k = n = 1000$) edges.

systems. Details of the experiments and results are in an appendix in Liu et al. (2026). Key findings from comparing and confirming the analyzed results with the experimental results are as follows:

1. Overall, precise calculation of optimal complexity and dedicated analysis of different inference and optimization methods in Section 4 help tremendously in predicting their performances and understanding the differences, for both asymptotic trends and constant factors, across not only the inference and optimization methods but also rule variants and graph types.
2. Which inference method and system to use depends on the unique features and powers the method and system provide, including the advantages summarized in Table 1. Whichever method and system are used, the rule variants and input graph types can affect the performance much more, as described below.
3. Efficient recursion types depend on the inference method and system. Except for an XSB performance bug discovered for the simplest case (left recursion on $\mathtt{W_{n,k}}$), described below, and relatively small XSB and Soufflé variations on the next simplest




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. 4: Running times on `BA` graphs, up to 100000 vertices, 200000 edges.

case (left recursion on $\texttt{X}_{\texttt{n,k}}$), (1) for XSB and clingo, left recursion is fastest; (2) for Soufflé, left and right recursions are about the same, faster than double recursion; and (3) overall, left recursion is fastest, double recursion can be asymptotically slower, all as analyzed in Section 4.

4. On all graphs except for left recursion on $\texttt{W}_{\texttt{n,k}}$ due to the performance bug, XSB, with query-driven inference with tabling, is the fastest, by several times or even an order of magnitude. clingo is the next fastest except for right recursion for `Cmpl` (e.g., Fig. 1) and `MaxAcyc` where Soufflé is faster. Soufflé is fastest in reading data and writing results (e.g., Fig. 3), even if it has the extra compilation time.
   We discovered the XSB performance bug when running on $\texttt{W}_{\texttt{n,k}}$ with increasing `k` = `n` (Fig. 3). This is the simplest case of all graph types, where `path` is exactly `edge` with no extra paths at all, but XSB on left recursion is drastically slower than analyzed. XSB team confirmed this performance bug and analyzed that it is due to hash collisions in this case.
5. Besides fixing the XSB bug, XSB and Clingo could be improved by automatically converting right recursion to left recursion (Tekle et al. 2008). Clingo's right and double recursions could perhaps be improved, possibly with better incremental processing for later hypotheses in a rule. Soufflé has opportunities to become faster in general, for all recursion types, because generated C++ code can be specialized to be much faster than general evaluator code (Rothamel and Liu 2007).

We also used well-known graph generators designed to simulate real-world networks and experimented with all three recursion types using all three rule systems. Fig. 4 shows the running times on directed Barabási-Albert (BA) graphs (Albert and Barabási 2002), the most famous model for explaining the scale-free nature of the Internet, World Wide Web, social media, and citation networks. The graphs are generated using the widely-used library NetworkX with preferential attachment parameter value 2.

The results in Fig. 4 are consistent with everything we have analyzed, in terms of relative performances of all recursion types and all systems, including a similar performance anomaly in left recursion in XSB as `W` graphs in Fig. 3. We think the anomaly is due to the same reason, because BA graphs are known to have ultra-small path lengths, with average length growing logarithmically with the number of vertices, whereas `W` graphs have only paths of length 1, so the anomaly for BA graphs is not as serious as for `W` graphs.

## 6 Related work and conclusion

There has been extensive study of efficiency of inference methods and systems, from performance benchmarking to complexity analysis. We discuss closely related works.

Prolog has had benchmarks for comparing performance of different implementations, e.g., (CMU AI Repository 1985; CHINA 2001). Some are focused on a special class of problems, e.g., interpreters written in Prolog (Körner et al. 2020). Some are for evaluating the performance of a particular implementation, e.g., SWI Prolog (SWI-Prolog bench 2026).

There are also works that evaluate queries in more general rule engines and database systems, e.g., the LUBM benchmark (Guo et al. 2005) and its extensions (Ma et al. 2006; Singh et al. 2020) for OWL on a range of systems, and evaluations including SQL with rules (Brass and Wenzel 2019a;b). In particular, rbench (Brass and Wenzel 2019b) considers 11 types of graphs, and we use all of them.

OpenRuleBench (Liang et al. 2009) was extensively constructed for comprehensively evaluating a wide range of systems on a diverse set of problems. Benchmarking (Liu et al. 2023b) for Alda (Liu et al. 2023a) used all OpenRuleBench benchmarks and more to evaluate an extension of DistAlgo (Liu et al. 2017)/Python with Datalog rules that implements queries using XSB, and to compare with programs written purely in Python, DistAlgo, and XSB.

More studies of methods and systems compare on some chosen workloads, e.g., Flan (Abeysinghe et al. 2024) on adding and compiling Datalog to Scala compares with Soufflé and two Datalog systems embedded in Rust, Crepe and Ascent, on program analysis; BMLP (Ai and Muggleton 2024) on using Boolean matrix operations for rules compares with Soufflé, clingo, B-Prolog, and SWI-Prolog on rules for graphs.

Datalog systems are increasingly studied (Maier et al. 2018; Vianu 2021; Ketsman et al. 2022), e.g., Ketsman et al. (2022) focuses on 9 Datalog systems including Soufflé, among 17 Datalog engines and more mentioned, and discusses the extensively-used semi-naive evaluation and more. There are also dedicated works on evaluating their performances, e.g., (Fan et al. 2022; Qureshi and Faber 2024). In particular, Fan et al. (2022) studies semi-naive execution profile, running time, CPU utilization, and memory consumption on transitive closure and four other closely related problems on binary relations.

None of the existing works analyzes and compares together all three widely different inference methods and their optimizations that make large performance impacts. Also, none of them analyzes different rule variants for the same problem and their significant performance differences. Furthermore, none of them studies precise optimal complexities, or performs detailed measurements from compiling and loading programs to writing query results, let alone confirming measurements against the complexities across inference methods, rule variants, and input graph types.

McAllester and Ganzinger (McAllester 1999; Ganzinger and McAllester 2002) analyze complexities of bottom-up evaluation using prefix firing counting. Liu and Stoller (Liu and Stoller 2003; 2009) develop systematic incrementalization to automatically transform Datalog rules into efficient implementations with precise time and space guarantees for optimality and trade-offs. Tekle and Liu (Tekle and Liu 2010; 2011) precisely analyze complexities of top-down evaluation for demand-driven queries and develop demand

transformations for bottom-up evaluation, improving over magic-set transformations exponentially in program size.

These studies of complexity analysis have been essential for the optimal complexity calculation in this paper, and for precisely understanding the efficiency of top-down query-driven evaluation with tabling vs. bottom-up fact-driven evaluation. Their extensions to handle unrestricted negation, quantification, and aggregation, following a powerful unified semantics for them (Liu and Stoller 2020; 2022), will be essential for critical applications that need those features.

**Conclusion, limitations, and future works.** We have presented a systematic study of efficiency of analysis of transitive relations using different inference methods for different rule variants and input relationship graphs. Despite the compelling results for better understanding and utilizing different rule systems, this work has several limitations, leaving many avenues for future work.

First, in the experiments, the degree of superlinearity and constant costs are not analyzed systematically; analyzing them systematically can help better understand and optimize rule systems.

Second, the study is only about time efficiency; space (Liu and Stoller 2009; Tekle and Liu 2010; 2011), including cache effect on large data, and other measures (Fan et al. 2022) are important too. Also important are additional system-specific aspects that affect performance, e.g., XSB has subsumptive tabling (which is faster in some cases than default tabling) (Swift and Warren 2012), indexing directives, many input/output functions (for efficiency in different cases), and more (Swift et al. 2022).

Third, comparative analysis of different methods for demand-driven queries, not inferring all facts, needs to be studied. Demand transformations (Tekle and Liu 2010; 2011) would help clingo and Soufflé; otherwise, goal-directed queries in XSB would perform drastically better. Magic sets in the intelligent grounder of DLV (Calimeri et al. 2017) and in Soufflé (Souffle 2026) are important for practical efficiency. Transforming into best recursion variants (Tekle et al. 2008) will also help significantly, as well as employing modular approaches, e.g, (Hu et al. 2022).

Finally, our comparative analysis is only for analyzing transitive relations; many more analysis problems can be added, especially those with negation and aggregation in existing rule systems (Sagonas et al. 1994; Jordan et al. 2016; Gebser et al. 2019) and even quantification, all unrestricted.

**Acknowledgment.** We thank Prof. David Warren for discussions about XSB performance and for confirming and analyzing the XSB performance bug we discovered. We also thank anonymous reviewers for their very helpful comments, especially for giving the small example at the end of Section 4.1. This work was supported in part by NSF under grant CCF-1954837.

## References

Abeysinghe, S., Xhebraj, A., and Rompf, T. 2024. Flan: An expressive and efficient Datalog compiler for program analysis. *Proceedings of the ACM on Programming Languages*, *8*, POPL, 2577–2609.

Ai, L. and Muggleton, S. H. 2024. Boolean matrix logic programming. *Computing Research Repository*, *arXiv:2408.10369*.

Albert, R. and Barabási, A.-L. 2002. Statistical mechanics of complex networks. *Reviews of modern physics*, *74*, 1, 47.

Bancilhon, F. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies* 1986, pp. 165–178. Springer.

Bellomarini, L., Sallinger, E., and Gottlob, G. 2018. The Vadalog system: Datalog-based reasoning for knowledge graphs. *Proceedings of the VLDB Endowment*, *11*, 9.

Brass, S. and Wenzel, M. A new benchmark database and an analysis of transitive closure runtimes. In *Post-Proceedings of the 32nd Workshop on (Constraint) Logic Programming* 2019a, pp. 3–18. `https://wi.hwtk.de/WLP2018/Papers/WLP_2018_paper_1.pdf`.

Brass, S. and Wenzel, M. Performance analysis and comparison of deductive systems and SQL databases. In *Proceedings of the 3rd International Workshop on the Resurgence of Datalog in Academia and Industry* 2019b, pp. 27–38. CEUR-WS.org.

Cai, J. and Paige, R. 1988. Program derivation by fixed point computation. *Science of Computer Programming*, *11*, 197–261.

Calimeri, F., Fuscà, D., Perri, S., and Zangari, J. 2017. I-DLV: The new intelligent grounder of DLV. *Intelligenza Artificiale*, *11*, 1, 5–20.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, *43*, 1, 20–74.

CHINA 2001. The CHINA Benchmark Suite. `http://www.cs.unipr.it/China/Benchmarks`. Accessed Apr. 22, 2025.

clingo 2026. clingo and gringo. `https://potassco.org/clingo`. Accessed Feb. 27, 2026.

CMU AI Repository 1985. Prolog Benchmarking Suites. `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/code/bench/0.html`. Accessed Feb. 27, 2026.

Fan, Z., Mallireddy, S., and Koutris, P. Towards better understanding of the performance and design of Datalog systems. In *Datalog 2.0: 4th International Workshop on the Resurgence of Datalog in Academia and Industry* 2022, pp. 166–180.

Flores-Montoya, A. and Schulte, E. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)* 2020, pp. 1075–1092.

Ganzinger, H. and McAllester, D. A. Logical algorithms. In *Proceedings of the 18th International Conference on Logic Programming* 2002, pp. 209–223. Springer.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, *19*, 1, 27–82.

Grädel, E. On transitive closure logic. In *International Workshop on Computer Science Logic* 1991, pp. 149–163. Springer.

Guo, Y., Pan, Z., and Heflin, J. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, *3*, 2-3, 158–182.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., Melo, G. D., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., and others 2021. Knowledge graphs. *ACM Computing Surveys*, *54*, 4, 1–37.

Hristova, K., Tekle, K. T., and Liu, Y. A. Efficient trust management policy analysis from rules. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* 2007, pp. 211–220. ACM Press.

Hu, P., Motik, B., and Horrocks, I. 2022. Modular materialisation of datalog programs. *Artificial Intelligence*, *308*, 103726.

Jordan, H., Scholz, B., and Subotić, P. Soufflé: On synthesis of program analyzers. In *Proceedings of the Intl. Conference on Computer Aided Verification* 2016, pp. 422–430. Springer.

Kaufmann, B., Leone, N., Perri, S., and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Magazine*, *37*, 3, 25–32.

Ketsman, B., Koutris, P., and others 2022. Modern Datalog engines. *Foundations and Trends® in Databases*, *12*, 1, 1–68.

Körner, P., Schneider, D., and Leuschel, M. On the performance of bytecode interpreters in Prolog. In *International Workshop on Functional and Constraint Logic Programming* 2020, pp. 41–56. Springer.

Li, N. and Mitchell, J. C. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages* 2003, pp. 58–73. Springer.

Liang, S., Fodor, P., Wan, H., and Kifer, M. OpenRuleBench: An analysis of the performance of rule engines. In *Proceedings of the 18th International Conference on World Wide Web* 2009, pp. 601–610. ACM Press.

Liu, Y. A. 2013. *Systematic Program Design: From Clarity to Efficiency*. Cambridge University Press.

Liu, Y. A., Idogun, J., Stoller, S. D., and Tong, Y. 2025 (revised May 2026). Efficiency of analysis of transitive relations using query-driven, ground-and-solve, and fact-driven inference. *Computing Research Repository*, *arXiv:2504.21291 [cs.DB]*.

Liu, Y. A. and Stoller, S. D. From Datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* 2003, pp. 172–183. ACM Press.

Liu, Y. A. and Stoller, S. D. 2009. From Datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, *31*, 6, 1–38.

Liu, Y. A. and Stoller, S. D. 2020. Founded semantics and constraint semantics of logic rules. *Journal of Logic and Computation*, *30*, 8, 1609–1638.

Liu, Y. A. and Stoller, S. D. 2022. Recursive rules with aggregation: A simple unified semantics. *Journal of Logic and Computation*, *32*, 8, 1659–1693.

Liu, Y. A., Stoller, S. D., and Lin, B. 2017. From clarity to efficiency for distributed algorithms. *ACM Transactions on Programming Languages and Systems*, *39*, 3, 12:1–12:41.

Liu, Y. A., Stoller, S. D., Tong, Y., and Lin, B. 2023a. Integrating logic rules with everything else, seamlessly. *Theory and Practice of Logic Programming*, *23*a, 4, 678–695.

Liu, Y. A., Stoller, S. D., Tong, Y., Lin, B., and Tekle, K. T. 2022. Programming with rules and everything else, seamlessly. *Computing Research Repository*, *arXiv:2205.15204 [cs.PL]*.

Liu, Y. A., Stoller, S. D., Tong, Y., and Tekle, K. T. Benchmarking for integrating logic rules with everything else. In *Proceedings of the 39th International Conference on Logic Programming (Technical Communications)* 2023b, pp. 12–26. Open Publishing Association.

Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. 2009. Declarative networking. *Communications of the ACM*, *52*, 11, 87–95.

Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., and Liu, S. Towards a complete OWL ontology benchmark. In *European Semantic Web Conference* 2006, pp. 125–139. Springer.

Maier, D., Tekle, K. T., Kifer, M., and Warren, D. S. Datalog: Concepts, history and outlook. In Kifer, M. and Liu, Y. A., editors, *Declarative Logic Programming: Theory, Systems, and Applications* 2018, chapter 1, pp. 3–120. ACM and Morgan & Claypool.

Marques-Silva, J. P. and Sakallah, K. A. 1999. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, *48*, 5, 506–521.

McAllester, D. A. On the complexity analysis of static analyses. In *Proceedings of the 6th International Static Analysis Symposium* 1999, pp. 312–329. Springer.

Qureshi, H. M. and Faber, W. 2024. Evaluating Datalog tools for meta-reasoning over OWL 2 QL. *Theory and Practice of Logic Programming*, *24*, 2, 368–393.

Rothamel, T. and Liu, Y. A. Efficient implementation of tuple pattern based retrieval. In *Proceedings of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation* 2007, pp. 81–90.

Sagonas, K., Swift, T., and Warren, D. S. XSB as an efficient deductive database engine. In *Proceedings of the 1994 ACM SIGMOD Intl. Conf. Management of Data* 1994, pp. 442–453.

Seo, J., Guo, S., and Lam, M. S. Socialite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering* 2013, pp. 278–289.

Singh, G., Bhatia, S., and Mutharaju, R. OWL2Bench: a benchmark for OWL 2 reasoners. In *International semantic web conference* 2020, pp. 81–96. Springer.

Smaragdakis, Y. and Balatsouras, G. 2015. Pointer analysis. *Foundations and Trends in Programming Languages*, *2*, 1, 1–69.

Souffle 2026. The Soufflé Project. `https://github.com/souffle-lang`. Accessed Feb. 27, 2026.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog*. MIT Press, 2nd edition.

SWI-Prolog bench 2026. SWI-Prolog benchmark suite. `https://github.com/SWI-Prolog/bench`. Accessed Feb. 27, 2026.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, *12*, 1-2, 157–187.

Swift, T., Warren, D. S., Sagonas, K., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Marques, R. F., Saha, D., Dawson, S., and Kifer, M. 2022. *The XSB System Version 5.0,x*. `http://xsb.sourceforge.net`. Latest release October 29, 2017.

Tamaki, H. and Sato, T. OLD resolution with tabulation. In *Proceedings of the 3rd International Conference on Logic Programming* 1986, volume 225 of *LNCS*, pp. 84–98. Springer.

Tekle, K. T., Hristova, K., and Liu, Y. A. Generating specialized rules and programs for demand-driven analysis. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology* 2008, volume 5140 of *LNCS*, pp. 346–361. Springer.

Tekle, K. T. and Liu, Y. A. Precise complexity analysis for efficient Datalog queries. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* 2010, pp. 35–44.

Tekle, K. T. and Liu, Y. A. More efficient Datalog queries: Subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* 2011, pp. 661–672.

Vianu, V. Datalog unchained. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* 2021, pp. 57–69.

## Appendix A Proofs for optimal number of combinations in Table 3

We give proofs for the formulas in Table 3 for optimal number of combinations for the recursive rule for all 3 recursion variants—left recursion (`path(X,Y), edge(Y,Z)`), right recursion (`edge(X,Y), path(Y,Z)`), and double recursion (`path(X,Y), path(Y,Z)`)—and all graph types. The proofs use the numbers of edges and paths, and of values of variables (`X`, `Y`, and `Z`) holding the vertices.

$\texttt{Cmpl}_\texttt{n}$, with edges `{(i,j): i in 1..n, j in 1..n}`

Given `edge` holds for all pairs of vertices. By base case rule of `path`, `path` holds for all of them too. Therefore, for all 3 recursion variants, all combinations of $n$ vertices for all three variables must be counted, yielding $n^3$ combinations, as shown in Table 3.

$\texttt{MaxAcyc}_\texttt{n}$, with edges `{(i,j): i in 1..n-1, j in i+1..n}`

Given `edge` holds for all vertex pairs $(i, j)$ where $i < j$. By base case rule of `path`, `path` holds for all of them too. Therefore, for all 3 recursion variants, for each vertex $i$ from 1 to $n$ for variable `Y`, there is a path or edge from each vertex from 1 to $i - 1$ for `X` and a path or edge to each of vertex $i + 1$ to $n$ for `Z`. So the number of combinations is $\sum_{i=1..n}(i-1)(n-i)$, yielding the closed form in Table 3.

$\texttt{Cyc}_\texttt{n}$, with edges `{(i,i+1): i in 1..n-1} + {(n,1)}`

Given `edge` holds for all vertex pairs $(i, i + 1)$ for $i$ from 1 to $n - 1$ and for pair $(n, 1)$. By transitivity, `path` holds for all pairs of vertices. For left (resp. right) recursion, for each `path` pair $(i, j)$, there is only one edge going from $j$ (resp. to $i$), and thus the number of combinations is the size of `path`, which is $n^2$, as shown in Table 3. For double recursion, the analysis is as for $\texttt{Cmpl}_\texttt{n}$, yielding $n^3$ in Table 3.

$\texttt{CycExtra}_\texttt{n,k}$, with edges `{(i,(i-1 + t*n/(k+1)) mod n + 1): i in 1..n, t in 1..k} +` $\texttt{Cyc}_\texttt{n}$

This is as $\texttt{Cyc}_\texttt{n}$ but adds edges from each vertex to $k$ vertices, so there are $n + nk$ edges total. Still, `path` holds for all pairs of vertices. For left recursion, for each `path` pair $(i, j)$, there are $k$ more edges going from $j$, and thus the number of combinations is $n^2 + n^2k$, as shown in Table 3. For right recursion, for each of $n + nk$ edges, there are paths to $n$ vertices, and thus the number of combinations is again $n^2 + n^2k$, as shown in Table 3. For double recursion, the analysis is the same as for $\texttt{Cyc}_\texttt{n,k}$.

$\texttt{Path}_\texttt{n}$, with edges `{(i,i+1): i in 1..n-1}`

Given `edge` holds for all vertex pairs $(i, i + 1)$ for $i = 1..n$. By transitivity, `path` holds for all pairs $(i, j)$ where $i < j$, so the number of `path` pairs is $\sum_{i=1..n}(n-i)$, which equals $\frac{1}{2}(n-1)(n-2)$. For left (resp. right) recursion, for each `path` pair $(i, j)$, there is one edge going from $j$ (resp. to $i$), and thus the number of combinations is the number of `path` pairs, i.e., $\frac{1}{2}(n-1)(n-2)$, as in Table 3. For double recursion, the calculation is the same as for $\texttt{MaxAcyc}_\texttt{n}$, yielding the closed form in Table 3.

$\texttt{PathDisj}_\texttt{n,k}$, with edges `{(i,i+k): i in 1..(n-1)*k}`

This is the same as $\texttt{Path}_\texttt{n}$, but multiplied by $k$ for each recursion variant.

$\texttt{Grid}_\texttt{n}$, with edges `{(j, j+1): i in 1..n, j in (i-1)*n+1..i*n-1} +`
`{(j, j+n): i in 1..n-1, j in (i-1)*n+1..i*n}`

For left recursion, for each vertex in position $(i, j)$ of the grid not in the last row or column, there are paths from $i \times j - 1$ vertices before and edges to 2 vertices after; for each vertex in the last column on row $i$ but not last row, there are paths from $i \times n - 1$ vertices and an edge to 1 vertex; for each vertex in the last row but on column j not last column, there are paths from $n \times j - 1$ vertices and an edge to 1 vertex. Thus, the number of combinations is $\sum_{i=1..n-1,j=1..n-1}(i \times j - 1) \times 2 + \sum_{i=1..n-1}(i \times n - 1) + \sum_{j=1..n-1}(n \times j - 1)$, yielding the closed form in Table 3.

For right recursion, the counting is similar, except to consider each vertex not in the first row or column, then each vertex in first column but not first row, and each vertex in first row but not first column, edges from 2 or 1 vertex before, and paths to vertices after. Thus, the number of combinations is $\sum_{i=2..n,j=2..n} 2 \times ((n - i + 1) \times (n - j + 1) - 1) + \sum_{i=2..n}((n - i + 1) \times n - 1) + \sum_{j=1..n}(n \times (n - j + 1) - 1)$, yielding the closed form in Table 3.

For double recursion, for each vertex in position $i = 1..n$ and $j = 1..n$ in the grid, there are paths from $i \times j - 1$ vertices before, and paths to $(n - i + 1) \times (n - j + 1) - 1$ vertices after. Thus the number of combinations is $\sum_{i=1..n,j=1..n}(i \times j - 1) \times ((n - i + 1) \times (n - j + 1) - 1)$, yielding the closed form in Table 3.

$\texttt{BinTree}_\texttt{h}$, with edges `{(i,2i): i in 1..2^h-1} + {(i,2i+1): i in 1..2^h-1}` and
$\texttt{BinTreeRev}_\texttt{h}$, with edges `{(2i,i): i in 1..2^h-1} + {(2i+1,i): i in 1..2^h-1}`

Left and right recursions are as proved in Section 4.1. Double recursion for $\texttt{BinTree}_\texttt{h}$ (resp. $\texttt{BinTreeRev}_\texttt{h}$) considers each level $i = 1..h$ from root to leaves, with $2^{i-1}$ vertices at level $i$, each having paths from (resp. to) $i - 1$ vertices, and to (resp. from) $2^{h-i+1} - 2$ vertices. Thus the number of combinations is $\sum_{i=1..h} 2^{i-1} \times (i - 1) \times (2^{h-i+1} - 2)$, yielding the closed form in Table 3.

$\texttt{X}_\texttt{n,k}$, with edges `{(i,n+1): i in 1..n} + {(n+1,n+1+j): j in 1..k}`

All 3 recursion variants just join the two sets of edges, and thus the number of combinations is $nk$, as shown in Table 3.

$\texttt{Y}_\texttt{n,k}$, with edges `{(i,n+1): i in 1..n} + {(i,i+1): i in n+1..n+k-1}`

For left recursion, each vertex $i$ from 1 to $k - 1$ in the stem of Y has paths from $n + i - 1$ vertices. Thus the number of combinations is $\sum_{i=1..k-1}(n + i - 1)$, yielding the closed form in Table 3.

For right recursion, vertex $n + 1$ has edges from $n$ vertices and paths to $k - 1$ vertices; each vertex $i$ from 2 to $k - 1$ in the stem of Y has edge from one vertex and paths to $k - i$ vertices. Thus the number of combinations is $n \times (k - 1) + \sum_{i=2..k}(k - i)$, yielding the closed form in Table 3.

For double recursion, each vertex $i$ as in left recursion has also paths from $n + i - 1$ vertices, but has paths to $k - i$ vertices. Thus, the number of combinations is $\sum_{i=1..k-1}(n + i - 1) \times (k - i)$, yielding the closed form in Table 3.

$\texttt{W}_\texttt{n,k}$, with edges `{(i,n+1 + (i+j-1) mod n): i in 1..n, j in 1..k}`

This graph has only paths of length 1. All 3 recursion variants have the number of combinations being 0, as shown in Table 3.

## Appendix B Performance measurements and analysis

We designed and performed detailed experiments with all three rule systems XSB, clingo, and Soufflé, all three rule variants, and all 12 input graph types.[1] We describe these experiments, analyze the measured running times, and compare with the analyzed optimal complexities.

For XSB, with the general programming power of Prolog, we wrote the scripting and timing code directly in XSB. We obtain separate times for (1) `LoadRules`—load rules (including `table` directive), using `consult`, (2) `ReadData`—read data from a file of facts, using `load_dync`, (3) `Query`—query all `path` facts, using `fail` to repeat without writing, and (4) `WriteRes`—write results to file, using `write`.

For clingo, with its interface for using it from Python, we wrote the scripting and timing code in Python. We obtain separate times for (1) `LoadRules`—load rules (including `show` directive), using `load`, (2) `ReadData`—read data from a file of facts, using `load`, (3) `Ground`—ground rules, using `ground`, and (4) `Solve`—search for all `path` facts, using `solve`, and (5) `WriteRes`—write results to file, using `write`.

For Soufflé, with its interface for using it from C++, we wrote the scripting and timing code in C++. We obtain separate times for (1) `RulesToC++`—compile rules (with type and input-output declarations) to C++, using `souffle` command line, (2) `C++ToExec`—compile C++ to executable, using `g++` command line, (3) `LoadRules`—load compiled executable for rules, using `newInstance`, (4) `ReadData`—read data from a directory of files of facts, one for each given predicate, using `loadAll`, (5) `Query`—compute all path facts, using `run`, and (6) `WriteRes`—write results to file in CSV format, using `printAll`.

For each input graph type, we generate graphs, exactly as defined in Table 2, with increasing values for `n` and `h`, and with both constant values and increasing values for `k`.

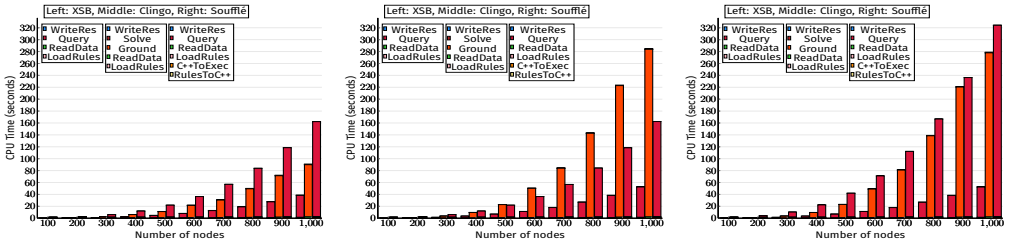
For each experiment, we report average CPU time over 5 runs. The variations across runs are negligible. All measurements were taken on an Apple M3 Pro chip, 2024, with a 4.05 GHz 11-Core CPU and 18 GB unified memory, 512 GB SSD, running MacOS Sonoma 14.5, XSB 5.0.0, clingo 5.7.1, and Soufflé 2.4.1-50-ge6cc66820.

### *B.1 Running times, confirmations, and exceptions*

We discuss running times first for graph types with a single parameter, `n` or `h`, and then for graph types with two parameters, `n` and `k`, but we compare across all graph types, and for all recursion variants and all systems.

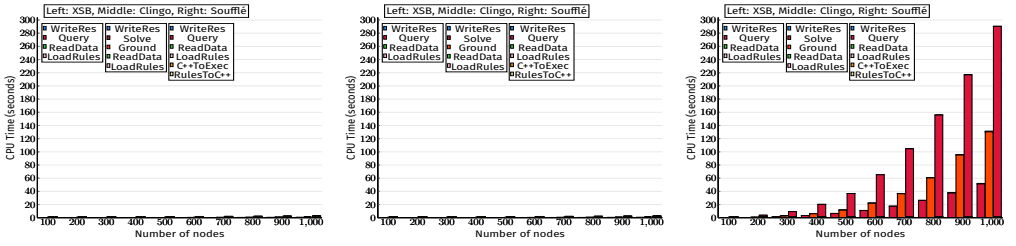
**Graph types with a single parameter.** We first examine the running times for five drastically different graph types with parameter `n` for number of vertices: $\texttt{Cmpl}_\texttt{n}$, $\texttt{Cycle}_\texttt{n}$, $\texttt{MaxAcyc}_\texttt{n}$, $\texttt{Path}_\texttt{n}$, and $\texttt{Grid}_\texttt{n}$, including densest (`Cmpl` graphs) and sparsest connected (`Path` graphs). We explain how the calculated precise complexities in Table 3 and the different optimized inference efficiencies in Section 4.2 are confirmed by the sameness and differences as well as trends and constants in actual inference times—`Query` for XSB

[1] Our benchmarking experiments are available at `https://github.com/Sirneij/trans-bench`.




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 1: Running times on `Cmpl` graphs, up to 1000 vertices, 1000000 edges.




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 2: Running times on `Cyc` graphs, up to 1000 vertices, 1000 edges.

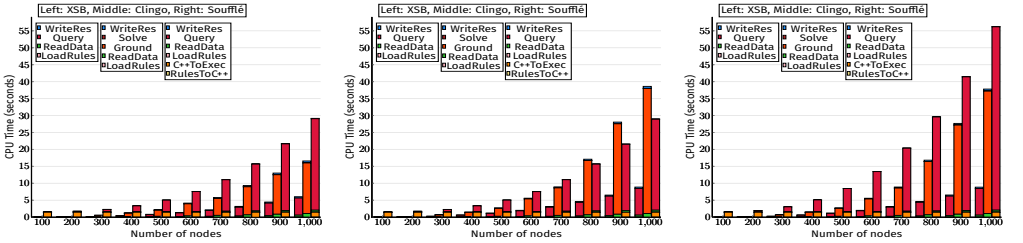
and Soufflé and `Ground` and `Solve` for clingo. The input and output times are also explained by data and result sizes, but they are relatively small here, so we focus on inference times.

Figs. B 1 to B 5 show the running time details for $\texttt{Cmpl}_{\texttt{n}}$, $\texttt{Cycle}_{\texttt{n}}$, $\texttt{MaxAcyc}_{\texttt{n}}$, $\texttt{Path}_{\texttt{n}}$, and $\texttt{Grid}_{\texttt{n}}$, for the number of vertices up to 1000 and the number of edges varying the mostly widely possible, from 1000000 (for $\texttt{Cmpl}_{\texttt{1000}}$) to 999 (for $\texttt{Path}_{\texttt{1000}}$) for a graph to stay connected. First, we observe the following that hold across all Figures, not just the five figures here—all consistent with the analysis in Section 4.2—except for an XSB performance bug discovered on the simplest case (`W` graphs, left recursion), described later:

- Soufflé has an extra compilation time (about 1.5 seconds, with tiny `RulesToC++`, in yellow, and virtually all `C++ToExec`, in orange). Of course, this time needs to be incurred only once for each program, not each run of the program.
- clingo's grounding time (`Ground`, in slightly lighter red) dominates its solving time (`Solve`) which is tiny, because optimized grounding already inferred all resulting facts.
- XSB is the fastest in all runs, confirming the overhead of tabling is minimal compared with the cost of grounding in clingo and semi-naive evaluation in Soufflé.

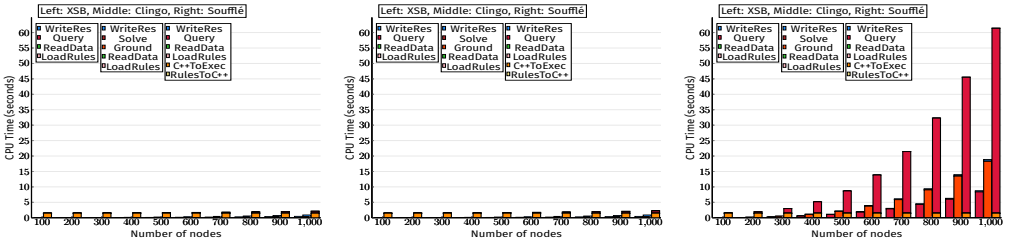
We also confirmed the trends of being superlinear in `n` for these graph types, as well as, for all other graph types, the expected trends of being linear or superlinear in `n` and `k` and of an explosive increase with respect to `h`, as analyzed in Table 3, except for the case where the XSB performance bug was discovered.

For more interesting detailed comparison among different recursions with different inference methods, we first examine Fig. B 1, for $\texttt{Cmpl}_{\texttt{n}}$. It is the most expensive case, with 1,000,000,000 combinations for 1000 vertices and 1,000,000 edges. Table 3 says that




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 3: Running times on `MaxAcyc` graphs, up to 1000 vertices, 499500 edges.




(a) Left recursion (b) Right recursion (c) Double recursion

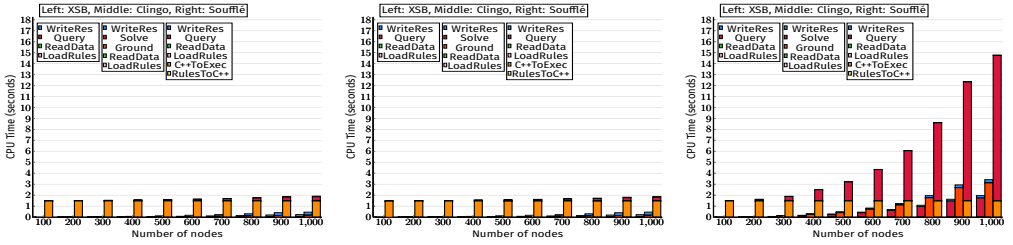
Fig. B 4: Running times on `Path` graphs, up to 1000 vertices, 999 edges.

all three recursion types have the same complexity, but Fig. B 1 shows it is only partially the case, and the difference confirms the analysis in Section 4.2:

- For both XSB and clingo, right and double recursions have about the same running times, but slower than left recursion (by about 40% for XSB and 220% for clingo). The exceptions are actually consistent with the analysis in Section 4.2: left recursion is faster because both consider hypotheses of rules from left to right, and clingo is slower because it uses ground rules and semi-naive evaluation.
- For Soufflé, left and right recursions have about the same running times, consistent with the analysis in Section 4.2, but double recursion is twice as slow. Indeed, with semi-naive, considering all new `path` facts in each iteration for double recursion has to essentially process each `path` fact twice, compared with only once for left or right recursion.

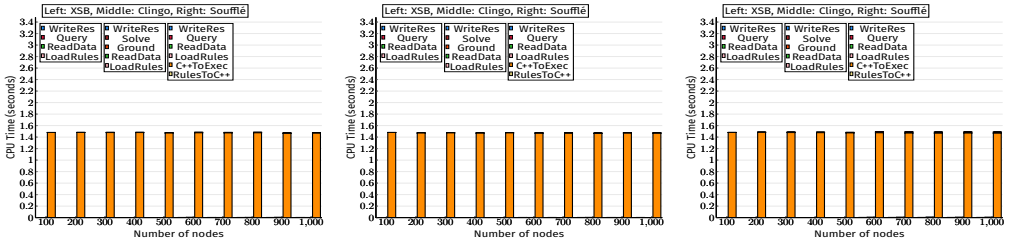
Examining Figs. B 2 to B 4 for other graph types shows and confirms additional details.

- For $\texttt{Cycle}_\texttt{n}$, Table 3 says that double recursion has the same complexity as $\texttt{Cmpl}_\texttt{n}$, whereas left and right recursions are a linear factor faster. Fig. B 2 indeed supports those for XSB and Soufflé, compared with Fig. B 1 for $\texttt{Cmpl}_\texttt{n}$. For clingo, it is faster in Fig. B 2(c) than in Fig. B 1(c), an effect of grounding over sparser input.
- For $\texttt{MaxAcyc}_\texttt{n}$, Table 3 says that all three recursion types have the same complexity, about 1/6 of that of $\texttt{Cmpl}_\texttt{n}$. Fig. B 3 confirms that for XSB and Soufflé, including the constant 1/6. For example, Soufflé on double recursion takes about 54 seconds (55.5-1.5) at 1000 vertices here, and about 322 seconds (323.5-1.5) in Fig. B 1. For clingo, slightly sparser graphs here have slightly less running times than in Fig. B 1.
- For $\texttt{Path}_\texttt{n}$, the relationship of Fig. B 4 to Fig. B 3 is exactly similar as the relationship of Fig. B 2 to Fig. B 1.



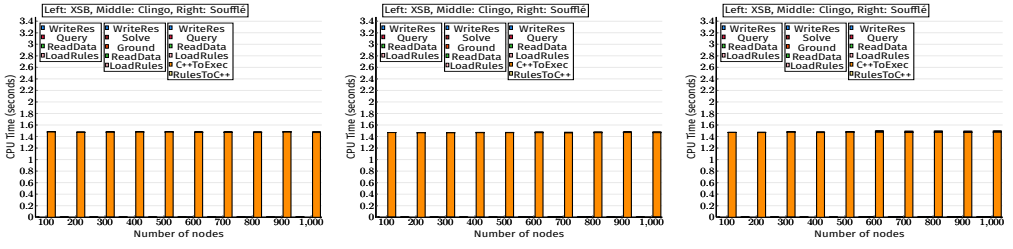

(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 5: Running times on `Grid` graphs, up to $\lfloor\sqrt{1000}\rfloor^2$ (= 961, closest from below to form a grid, and calculated this way for all x-axis labels) vertices, 1860 edges.




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 6: Running times on `BinTree` graphs, up to $2^{\lfloor\log_2 1000\rfloor}-1$ (= 511, closest from below to 1000 to form a binary tree, and calculated this way for all x-axis labels) vertices, 510 edges.

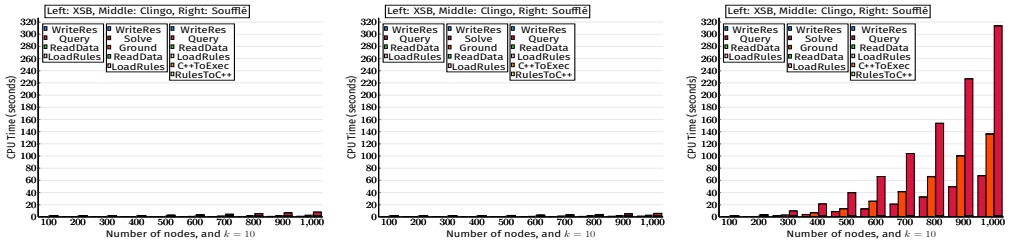



(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 7: Running times on `BinTreeRev` graphs, up to $2^{\lfloor\log_2 1000\rfloor}-1$ vertices, 510 edges.

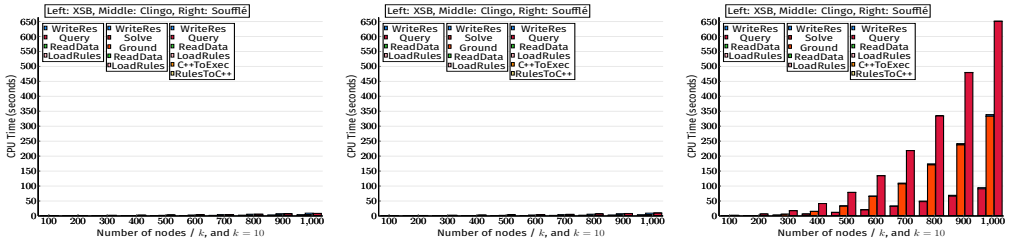
Similar analysis as above applies to $\texttt{Grid}_{\texttt{n}}$, where `n` is (the closest from below to) the square root of the number of vertices. The running times are shown in Fig. B 5. The main difference from first four graph types is that the inference times are noticeably smaller, and that other times besides inference times are more visible. This is due to much shorter paths to follow (shorter than the square root of `n`).

Similar analysis applies also to $\texttt{BinTree}_{\texttt{h}}$ and $\texttt{BinTreeRev}_{\texttt{h}}$ with parameter `h` for level of nodes, except that paths are even drastically shorter (logarithm of the number of nodes). The running times, shown in Figs. B 6 and B 7, are so small that only the compilation time shows distinctively. We also experimented with trees with drastically more nodes and observed that the running times increase explosively with respect to `h`, and that both tree types have about the same running times, all consistent with analyzed.




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 8: Running times on `CycExtra` graphs, up to 1000 vertices, 11000 edges.



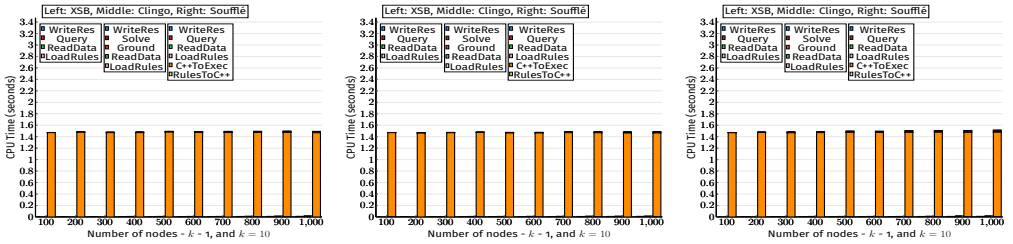

(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 9: Running times on `PathDisj` graphs, up to 10000 vertices, 9990 edges.

**Graph types with two parameters.** The remaining five graph types—$\texttt{CycExtra}_{\texttt{n,k}}$, $\texttt{PathDisj}_{\texttt{n,k}}$, $\texttt{X}_{\texttt{n,k}}$, $\texttt{Y}_{\texttt{n,k}}$, and $\texttt{W}_{\texttt{n,k}}$—have two parameters `n` and `k`. We discuss results with two sets of experiments, both with increasing `n` up to 1000: (1) with constant $\texttt{k} = 10$ for all five graph types, and (2) with increasing $\texttt{k} = \texttt{n}$ for the last three graph types. The latter helps greatly in showing the different impacts on performance by the last three graph types. It also led us discover an XSB performance bug in the simplest case of left recursion on `W` graphs.

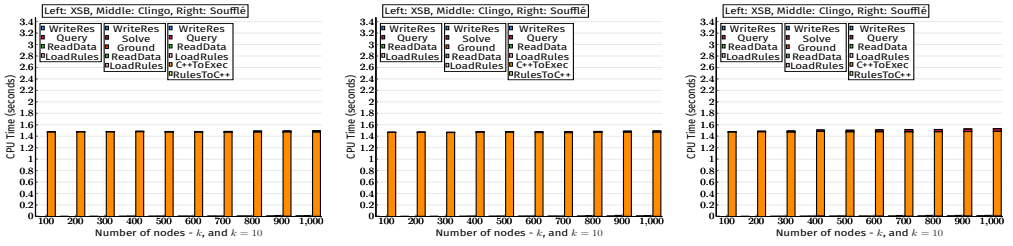
Figs. B 8 to B 12 show the running times for all five kinds of graphs for constant $\texttt{k} = 0$. They confirm the analysis as before and additionally the constant factor differences except for what we believe is a good cache locality.

- Comparing Fig. B 8 for $\texttt{CycExtra}_{\texttt{n,10}}$ with Fig. B 2 for $\texttt{Cyc}_{\texttt{n}}$, we can see they are similar for double recursion, consistent with the analysis in Section 4. Running times for left and right recursions are too small in these figures, but they generally are about 4-5 times for XSB and Soufflé and about 3 times for clingo (e.g., from 0.10 to 0.45 seconds for XSB, from 0.66 to 1.85 seconds for clingo, and from 1.28 to 6.17 seconds for Soufflé, for left recursion on 1000 vertices).
  These constant factors are smaller than the 10-times-larger analyzed in Section 4. We believe this is due to the good cache locality of 10 added edges starting from a same vertex in $\texttt{CycExtra}_{\texttt{n,10}}$, contrasting each edge starting from a different vertex in $\texttt{Cyc}_{\texttt{n}}$.
- Comparing Fig. B 9 for $\texttt{PathDisj}_{\texttt{n,10}}$ with Fig. B 4 for $\texttt{Path}_{\texttt{n}}$, we can clearly see that the running times are about 10 times as large for double recursion, again with analysis in Section 4. For left and right recursions, this is true too even though the running times are again too small to see in the figures (e.g., from 0.05 to 0.59



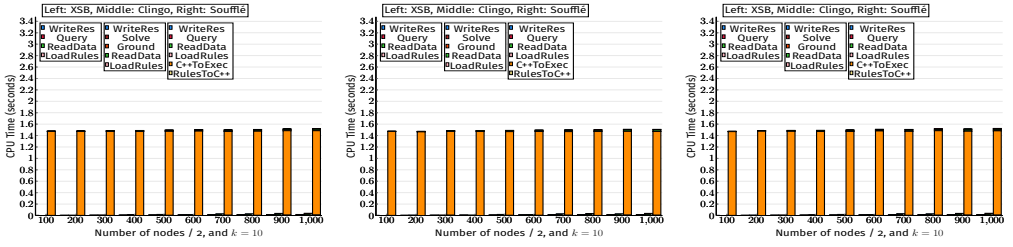

(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 10: Running times on `X` graphs, up to 1011 vertices, 1010 edges.




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 11: Running times on `Y` graphs, up to 1010 vertices, 1009 edges.




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 12: Running times on `w` graphs, up to 2000 vertices, 10000 edges.

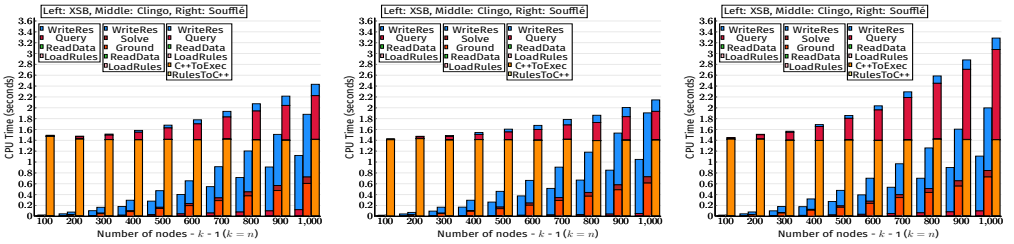
seconds for XSB, from 0.33 to 3.78 seconds for clingo, and from 0.60 to 6.31 seconds for Soufflé). Indeed, each $\texttt{PathDisj}_{\texttt{n,10}}$ graph is exactly the same as 10 $\texttt{Path}_{\texttt{n}}$ graphs.

- Figures for the last three graph types show that the running times for `k` = 10 are all so small that the compilation time stands out again. But those small running times do show different coloring, prompting the next set of experiments.

Figs. B 13 to B 15 show the running times for the last three graph types for `k` = `n`, i.e., `k` increases as `n` increases. Fig. B 14 for $\texttt{Y}_{\texttt{n,n}}$ can be analyzed similarly as for $\texttt{Path}_{\texttt{n}}$, but the most interesting cases are Figs. B 13 and B 15 for $\texttt{X}_{\texttt{n,k}}$ and $\texttt{W}_{\texttt{n,k}}$, respectively.

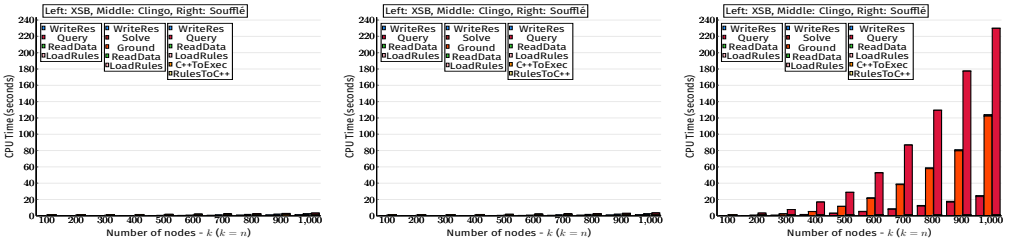
- Fig. B 13 for $\texttt{X}_{\texttt{n,n}}$ shows two new aspects clearly: (1) `WriteRes` times (in blue) are totally obvious, and they are much larger for XSB and clingo than for Soufflé; and (2) `Solve` times (in darker red) in clingo are obvious on top of `Ground` times (in lighter red).

  Fig. B 13 also shows an exception from before: inference time for left recursion got worse, relative to right recursion, for all of XSB, clingo, and Soufflé. XSB is about



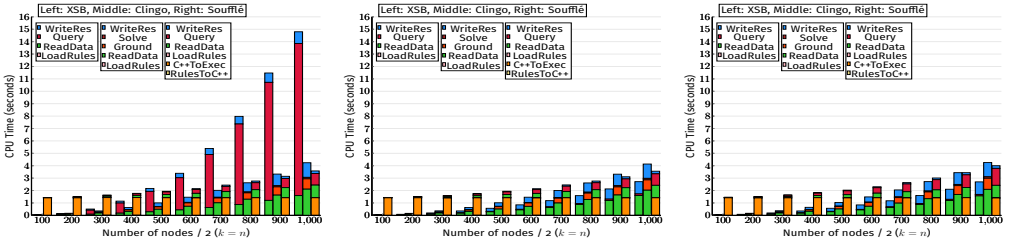

(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 13: Running times on `X` graphs, up to 2001 vertices, 2000 edges.




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 14: Running times on `Y` graphs, up to 2000 vertices, 1999 edges.




(a) Left recursion (b) Right recursion (c) Double recursion

Fig. B 15: Running times on `W` graphs, up to 2000 vertices, 1000000 edges.

150% slower, instead of faster; clingo is about 1% faster, instead of much faster; Soufflé is about 50% slower, instead of the same. Note though that these variations are relatively very small compared with the times for writing results by XSB and clingo and compilation by Soufflé.

The exception is due to the specifics of `X` graphs having only paths of lengths 1 and 2, where faster table lookups for `path` pairs in left recursion is overwhelmed by the sheer number of them that fail anyway. In particular, $\texttt{X}_{\texttt{n,n}}$ has `#edge` = `2n` and `#path` = $\texttt{2n+n}^2$. Since XSB and clingo consider hypotheses from left to right, left recursion looks up $\texttt{2n+n}^2$ `path` pairs first, and only `n` of them succeed, whereas right recursion looks up `2n edge` pairs first, and `n` of them succeed. We believe this difference in the number of lookups is also the case for Soufflé. Optimal computation with minimum increment can achieve the same efficiency for all recursion types.

- Fig. B 15 for $\texttt{W}_{\texttt{n,n}}$ shows one more new aspect: `ReadData` times (in green) are totally obvious, and they are about twice as high for XSB and clingo than for Soufflé.

The biggest surprise is the discovery of an XSB performance bug: The running times of XSB on left recursion grow drastically higher than clingo and Soufflé and than analyzed. This is actually the simplest case among all graph types we consider, where `path` is exactly `edge` with no other paths at all.
This performance bug was confirmed by the XSB team. They then analyzed and determined that it is due to hash collisions in this case. They also showed that additional, more sophisticated indexing directives in XSB could be added to the program to bypass this bug.