Precise Complexity Analysis for Efficient Datalog Queries *

K. Tuncay Tekle Yanhong A. Liu
Department of Computer Science, State University of New York at Stony Brook
tuncay,liu@cs.sunysb.edu

Abstract

Given a set of Datalog rules, facts, and a query, answers to the query can be inferred bottom-up starting with the facts or top-down starting with the query. For efficiently answering the query, top-down evaluation is extended with tabling that stores the results of the subqueries encountered, and bottom-up evaluation is done on rules transformed based on demand from the query.

This paper describes precise time and space complexity analysis for efficiently answering Datalog queries, and precise relationships between top-down evaluation with tabling and bottom-up evaluation driven by demand. We first present a systematic method for precisely calculating the worst-case time and space complexities of top-down evaluation with tabling. We then describe a method for transforming the rules for efficiently answering queries using bottom-up evaluation of the transformed rules; the method is akin to the magic set transformation, but is simpler and produces simpler rules that yield exponentially smaller space in the number of arguments of predicates. Next, we establish precise relationships between top-down evaluation with tabling and bottom-up evaluation of rules transformed based on demand. Finally, we support our analyses and comparisons through experiments on benchmarks from OpenRuleBench.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—constraint and logic languages; D.3.4 [*Programming Languages*]: Processors—Optimization; F.2.2 [*Analysis of Algorithms and Problems Complexity*]: Nonnumerical Algorithms and Problems—Computations on discrete structures; H.2.3 [*Information Systems*]: Database Management—Query languages; H.2.4 [*Information Systems*]: Systems—Query processing, Rule-based databases

General Terms Languages, Performance

Keywords Complexity analysis, Datalog, demand-driven evaluation, program transformation, optimization, tabling

1. Introduction

Datalog [7] is a logic language used in deductive databases [1], program analysis [34], security [12], and many other applications [15, 17, 26]. Given a set of Datalog rules, facts, and a query, answers to the query can be inferred using bottom-up evaluation starting with the facts or top-down evaluation starting with the query. Many evaluation methods have been studied [2, 6], including [16, 31, 32], and notably top-down evaluation with tabling [28] that guarantees termination in polynomial time, and optimal bottom-up evaluation with complexity guarantees [21] after program transformations such as the magic set transformation [3]; we refer to these evaluation methods as *tabled top-down evaluation* and *demand-driven bottom-up evaluation*, respectively.

Despite extensive research on improving Datalog evaluation methods, and on optimizing Datalog programs, e.g., [5, 8, 11, 22, 27], the performance of rule engines remains little understood [18, 19]. In particular, performance differences using different evaluation methods are most often drastic, and even using the same evaluation method, changing the order of hypotheses in rules most often yields dramatically different performance that is easily observed to be asymptotic. Recent work studied efficient bottom-up evaluation with precise complexity guarantees [21], but precise complexities for efficiently answering queries using tabled top-down evaluation remain unknown. There was significant research relating various top-down and bottom-up evaluation methods, as discussed in the related work section, but a large gap remains in precisely relating tabled top-down and demand-driven bottom-up evaluations.

This paper describes precise time and space complexity analysis for efficiently answering Datalog queries, and precise relationships between tabled top-down and demand-driven bottom-up evaluations. We first present a systematic method for precisely calculating the worst-case time and space complexities of tabled top-down evaluation. The calculation is based on possible binding patterns of arguments of predicates during the evaluation, and expresses the complexities in terms of parameters that characterize the actual number of facts used. We then describe *demand transformation*, which transforms Datalog rules for efficiently answering queries using bottom-up evaluation of the transformed rules. The transformation is akin to the magic set transformation, but is simpler and produces simpler rules that yield exponentially smaller space in the number of arguments of predicates.

Additionally, we establish precise relationships between tabled top-down evaluation and demand-driven bottom-up evaluation, in terms of precise time and space complexities. We show that the time complexity of demand-driven bottom-up evaluation is better than or equal to tabled top-down evaluation, and that for rules that have no more than two hypotheses and no wildcards, their complexities are equivalent. Then, we show that the space complexity of tabled top-down evaluation is better than or equal to demand-driven bottom-up evaluation, and that if the time complexity of demand-driven bottom-up evaluation is better than tabled top-down evaluation, then its space complexity must be worse. We have implemented our methods and confirmed our analysis results through experiments on benchmarks from OpenRuleBench [18].

* This work was supported in part by NSF under grants CCF-0964196 and CCF-061391 and by ONR under grant N000140910651.

PPDP'10, July 26–28, 2010, Hagenberg, Austria.

The rest of the paper is organized as follows. Section 2 presents Datalog and the terminology used in the paper. Section 3 gives the algorithm for tabled top-down evaluation and describes our method for precisely calculating the complexities. Section 4 describes demand transformation for demand-driven bottom-up evaluation and compares the transformation with the magic set transformation. Section 5 establishes the relationship between tabled top-down and demand-driven bottom-up evaluations. Section 6 presents results from experiments. Section 7 discusses related work and concludes.

2. Datalog

Datalog is a language for defining rules, facts, and queries, where rules can be used with facts to answer queries. A Datalog rule is of the form:

$$p(a_1, ..., a_k) :- p_1(a_{11}, ..., a_{1k_1}), ..., \ p_h(a_{h1}, ..., a_{hk_h}).$$

where h is a finite natural number, each p_i (respectively p) is a predicate of finite number k_i (respectively k) arguments, each a_{ij} and a_i is either a constant or a variable, and each variable in the arguments of p must also be in the arguments of some p_i.

A predicate with arguments is called an *atom*. If the right side of a rule is empty, then the atom on the left must have only constant arguments, and is called a *fact*; we indicate a fact with an ending dot. If the left side of a rule is empty, then each atom on the right side is called a *query*; we indicate a query with an ending question mark. For the rest of the paper, "rule" refers only to the case where both sides of the rule are not empty, where each atom on the right is called a *hypothesis*, and the atom on the left is called the *conclusion*.

The meaning of a set of rules, facts, and queries is the set of facts that are given or can be inferred using the rules and that match the queries. We will consider the case of one query, because a set of queries is equivalent to one query by adding a rule whose hypotheses are the set of queries and whose conclusion is the one query.

Terminology. An *IDB (intensional database) predicate* is a predicate defined by rules. An *IDB hypothesis* is a hypothesis whose predicates is an IDB predicate.

In examples, we use `x`, `y`, and `z` for variables, and we use `c` for constants.

For complexity calculation, we use the following notations.

- `#p`: the number of facts of predicate `p`, called the *size of* `p`.
- `#p.`$\mathtt{i}_1$`,...,`$\mathtt{i}_n$`/`$\mathtt{j}_1$`,...,`$\mathtt{j}_m$: the maximum number of combinations of different values of the $\mathtt{i}_1$,...,$\mathtt{i}_n$th arguments of the facts of predicate `p` (given or inferred), given any fixed value for the $\mathtt{j}_1$,...,$\mathtt{j}_m$th arguments.
- `#p.i`: *actual* number of values of the ith argument of a particular instance of `p`.
- `dom(p.i)`: the size of the domain of the `i`th argument of predicate `p`, i.e., the number of all possible values of that argument of `p`.

Running examples. We use two versions of transitive closure as running examples. The first is left-recursive, defined using a base-case rule `(B)` and a left-recursive rule `(L)`:

```
path(x,y) :- edge(x,y).                (B)
path(x,y) :- path(x,z), edge(z,y).     (L)
```

The second is right-recursive, defined using the same base-case rule `(B)`, and a right-recursive rule `(R)`:

```
path(x,y) :- edge(x,z), path(z,y).     (R)
```

We consider two queries, `path(c,y)?` and `path(x,c)?` that find all targets reachable from a constant `c`, and all sources that reach a constant `c`, respectively. We call them target query and source query, respectively.

3. Complexity analysis for tabled top-down evaluation

To answer a query, top-down evaluation starts with the query, generates subqueries from hypotheses of rules whose conclusions match the query, considering rules in the order given, and considering hypotheses from left to right, and does so repeatedly until the subqueries match given facts. This may lead to repeated subqueries or infinite recursion when recursive rules exist. To address this problem, *tabling* memoizes answers to subqueries, and reuses them when possible.

We consider top-down evaluation using variant tabling with depth-first scheduling and without early completion.

- *Variant tabling* [9] is the dominant tabling strategy. It stores and reuses the answers to *variant*s of previously encountered subqueries, where a subquery is a variant of another if they are equal modulo variable renaming.
- *Depth-first scheduling* selects the next subqueries to evaluate in a depth-first manner. The two major scheduling strategies, local and batched [13], have the same asymptotic time and space complexities as depth-first scheduling. We describe complexity analysis using depth-first scheduling, because it is simpler.
- *Early completion* stops evaluation for a subquery whose arguments are all bound, once the subquery is evaluated to be true. *No early completion* means using all relevant rules to infer answers to a subquery even if it is a subquery whose arguments are all bound and has been evaluated to be true.

We also make the following two assumptions.

- All IDB predicates are tabled. This allows the best possible asymptotic time complexity; it may use unnecessarily large space, which is a problem that should be addressed, but is beyond the scope of this paper.
- All predicates are perfectly indexed, so that it takes constant time to retrieve a fact of the predicate given fixed values for some of its arguments. In systems implementing tabled top-down evaluation, perfect indexing can be manually specified, such as in XSB [25], or is automatically performed, such as in YAP [10].

For the rest of the paper, *tabled top-down evaluation* refers to evaluation using variant tabling, with depth-first scheduling, without early completion, and with the two assumptions above.

Figure 1 gives the algorithm for tabled top-down evaluation. It recursively calls **invoke** as described below. Two global maps are used: $Table$ and $Suspension$. A map maps a key to a set of values, where each pair of key and set of values is called an entry. $Table$ maps each subquery encountered that is not a variant of a previously encountered subquery to a set of facts inferred for the subquery. The keys of $Suspension$ are pairs of atoms consisting of a key k of $Table$ and a hypothesis for which an answer for k can be used to resume computation. The values for each key are tuples of arguments to call **invoke** with when a fact for the hypothesis in the key is inferred.

In the algorithm, the following functions are used:

- **concl**(r) and **hypos**(r): the conclusion and the set of hypotheses of rule r, respectively.
- **unify**(a,b): a most general unifier of atoms a and b if it exists, $\emptyset$ otherwise.

- **subst**(a,θ): the atom a after substitution of variables using θ.
- **variant**(a,b): whether atoms a and b are equal modulo variable renaming.
- **keys**(m): keys of map m.

The algorithm starts from the given query, and calls procedure **invoke** for each rule whose conclusion matches the given query. The procedure takes four arguments:

1. a query q,
2. a rule r whose conclusion matches q,
3. an index i of the hypothesis of r to process,
4. a substitution θ from matching q against the conclusion of r, and matching facts against up to the ith hypothesis of r.

If the number of hypotheses of r is smaller than or equal to i, the procedure substitutes variables of the ith hypothesis of r using θ (called h_i), and performs the following on h_i:

- If h_i is not an IDB hypothesis, then find each fact that matches the hypothesis, and call **invoke** with i incremented for the next hypothesis, and with θ extended with the new match.
- If h_i is an IDB hypothesis and is a variant of an existing key of $Table$, then for each fact in the values for that key, match the fact against h_i, and call **invoke** with i incremented for the next hypothesis, and with θ extended with the new match. Also, record the current arguments of **invoke** for resuming computation after a new fact is added to the values of this table entry.
- If h_i is an IDB hypothesis and is not a variant of an existing table key, create a table entry whose key is h_i, and whose set of values is the empty set. For each rule r' whose conclusion matches h_i, call **invoke** with the arguments h_i, r', 1, and the substitution from the match. Also, record the current arguments of **invoke** for resuming computation after a new fact is added to the values for the new table entry.

If the number of hypotheses of r is smaller than i, then a fact is inferred and the substitution θ must contain all variables in q, because the rules are safe. The fact inferred is q after substitution using θ. The fact is added to the values for key q if it is not already in the values, and finally for each tuple of arguments that can resume computation with a new fact, **invoke** is called with the arguments after updating the substitution in the tuple to account for the inferred fact.

The time complexity is the number of calls to **invoke**, because all other operations are constant time in data size. We make the following observations for counting the number of calls to **invoke**: (1) The combination of the first two arguments of **invoke** are determined by the call to **invoke** whose index argument is 1, because other calls copy these two arguments from the enclosing call to **invoke**. (2) The calls to **invoke** whose index argument is 1 must be for queries that are not variants of the subqueries in $Table$, and match the conclusion of some rule. (3) For each pair of the first two arguments to **invoke**, rule r and query q that matches the conclusion of r, the combinations of the last two arguments, index i and substitution θ, are the combinations of facts that match the hypotheses of r.

The space complexity is the number of facts stored in the table entries. We do not consider stack space in this paper.

For easier and more precise calculation, we first generate a query and rules annotated with the patterns of argument bindings based on the given query, but whose evaluation is otherwise the same as the given query and rules. Then, we calculate the complexity of evaluating the annotated query and rules. Annotations make complexity calculation easier by distributing the complexity to parts of the query and rules that contribute to it in simpler ways.

```
Suspension = new map
Table = new map
Table[q] = ∅
// Call invoke for each rule matching q
for r ∈ R | θ = unify(concl(r), q) ≠ ∅:
  invoke(q, r, 1, θ)
return Table[q]

procedure invoke(q, r, i, θ):
  // If there are still hypotheses of r to process
  if i ≤ |hypos(r)|:
    h_i = subst(the ith hypothesis of r, θ)
    if h_i is not an IDB hypothesis:
      // Call invoke for each matching fact
      for fact ∈ F | θ' = unify(h_i, fact) ≠ ∅:
        invoke(q, r, i + 1, θ ∪ θ')
    // If h_i is a variant of an existing table key
    else if ∃k ∈ keys(Table) | variant(h_i, k):
      // Record current arguments
      // for resuming invoke later
      Suspension[⟨k, h_i⟩] ∪= {⟨q, r, θ, i⟩}
      // Call invoke for each fact in values for key k
      for fact ∈ Table[k]:
        θ' = unify(h_i, fact)
        invoke(q, r, i + 1, θ ∪ θ')
    // If a variant does not exist in table keys
    else:
      Table[h_i] = ∅
      // Record current arguments
      // for resuming invoke later
      Suspension[⟨h_i, h_i⟩] ∪= {⟨q, r, θ, i⟩}
      // Call invoke for each r matching new query h_i
      for r' ∈ R | θ' = unify(concl(r'), h_i) ≠ ∅:
        invoke(h_i, r', 1, θ')
  // If no more hypothesis is left to process
  else:
    fact = subst(q, θ)
    // If the fact has not been inferred before
    if fact ∉ Table[q]:
      // Add the fact to the table
      Table[q] ∪= {fact}
      // Resume computations
      for ⟨k, h⟩ ∈ keys(Suspension) | k = q:
        for ⟨q', r', θ', i'⟩ ∈ Suspension[⟨q, h⟩]:
          θ'' = unify(h, fact)
          invoke(q', r', i' + 1, θ' ∪ θ'')
endproc
```

Figure 1. Tabled top-down evaluation of query q, given a set of facts F and a set of rules R

3.1 Binding annotation

To annotate a set of rules with respect to a query, we first determine the patterns of argument bindings during the evaluation of the query, called *demand patterns*, and then generate an annotated rule for each pattern determined.

Demand patterns. Given a set of rules and a query, each subquery $\mathtt{p(a_1,\ldots,a_k)}$ encountered during tabled top-down evaluation yields a *demand pattern* $\langle \mathtt{p}, \mathtt{s} \rangle$, where $\mathtt{s}$ is a string, called the *pattern string*, of length k whose ith character is '$\mathtt{b}$' if $\mathtt{a}_i$ is bound,

and 'f' otherwise. For an atom $\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ and a pattern string s of length k, we say that $\texttt{a}_i$ is *bound by* s if the ith character of s is 'b'.

Demand patterns are computed iteratively as follows until no new demand patterns can be added. The demand pattern of the given query $\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ is $\langle \texttt{p}, \texttt{s} \rangle$, where the ith character of s is 'b' if $\texttt{a}_i$ is a constant, and 'f' otherwise. For each computed demand pattern $\langle \texttt{p}, \texttt{s} \rangle$, for each rule r that defines p, and for each IDB hypothesis h of r whose predicate is, say, q, add a demand pattern $\langle \texttt{q}, \texttt{t} \rangle$, where the ith character of t is 'b' if the ith argument of h is a constant, or appears in a hypothesis to the left of h in r, or is an argument of the conclusion of r bound by s; and 'f' otherwise.

Annotation. For each demand pattern $\langle \texttt{p}, \texttt{s} \rangle$ computed, and for each rule r that defines p, we generate an annotated rule that obeys the pattern string s, where the conclusion is annotated with s, and each hypothesis is annotated with the pattern string obtained as described above.

Formally, for each demand pattern $\langle \texttt{p}, \texttt{s} \rangle$, and each rule of the form

$\texttt{p(...) :- h}_1\texttt{(...),...,h}_n\texttt{(...).}$

We generate the rule

$\texttt{p\_s(...) :- h}_1\texttt{\_s}_1\texttt{(...),...,h}_n\texttt{\_s}_n\texttt{(...).}$

where for each $1 \leq k \leq n$, the ith character of $\texttt{s}_k$ is 'b' if the ith argument of $\texttt{h}_k$ is a constant, or appears in a hypothesis to the left of $\texttt{h}_k$, or is an argument of the conclusion bound by s, and 'f' otherwise.

For the given query p(...)?, the annotated query p_s(...)? is generated, where the ith character of s is 'b' if the ith argument of the given query is a constant; and 'f' otherwise.

Example. For rules (B) and (L), and target query path(c,y)?, the set of demand patterns is $\{\langle \texttt{path}, \texttt{'bf'} \rangle\}$, and annotation results in the annotated query path_bf(c,y)? and two annotated rules:

```
path_bf(x,y) :- edge_bf(x,y).                (B')
path_bf(x,y) :- path_bf(x,z), edge_bf(z,y).  (L')
```

For rules (B) and (R), and the same target query, the set of demand patterns is the same, and annotation results in the same annotated query, rule (B'), and the following rule:

```
path_bf(x,y) :- edge_bf(x,z), path_bf(z,y).  (R')
```

Annotation is the same as predicate splitting [30], except we annotate all hypotheses, in contrast to only IDB, for ease of complexity analyses.

3.2 Time complexity analysis

For an annotated rule, the asymptotic time complexity it incurs is the product of: (1) *local complexity*—the number of different values that the free variables in the rule can take, and (2) *number of invocations*—the number of different values that the bound arguments of the conclusion can take. We give a method to calculate an upper bound for each factor. Summing the complexities incurred by all rules gives the overall complexity.

The local complexity of a rule is the product of complexity factors incurred by all hypotheses of the rule. Each hypothesis, say $\texttt{p\_s(a}_1\texttt{,...,a}_n\texttt{)}$, of r incurs the complexity factor $O(\texttt{\#p.f}_1\texttt{,...,f}_k\texttt{/b}_1\texttt{,...,b}_l)$, where $\texttt{f}_i$ is the index of the ith 'f' in s, and $\texttt{b}_i$ is the index of the ith 'b' in s.

For example, for rule (L'), the first hypothesis incurs the complexity factor $O(\texttt{\#path.2/1})$, and the second hypothesis incurs the complexity factor $O(\texttt{\#edge.2/1})$. Therefore, the local complexity is $O(\texttt{\#path.2/1} \times \texttt{\#edge.2/1})$.

For computing the number of invocations of a rule r, three steps are performed. First, among all hypotheses of all rules and the given query, find those whose predicate is the same as the predicate of the conclusion of r. Second, for each one found, say called h, calculate the number of different values its bound arguments can take. If a bound argument is a constant, then it can take only that one value. If a bound argument is a variable, say x, then the minimum of the following is taken: (1) If x is the ith argument of a hypothesis to the left of h whose predicate is p, then x may take $O(\texttt{\#p.i})$ different values. (2) If x appears in the conclusion c, there are two cases: if c is a variant of h, and the bound arguments of c and h are the same, then x may take one value; otherwise it may take $O(\texttt{dom(p.i)})$ values, where p is the predicate of c, and x is the ith argument of c. The product of the numbers of different values that the bound arguments can take in h is the total number of invocations of r due to h. Third, the sum of the products due to all h's is the number of invocations to r.

For example, the predicate of the conclusion of rule (R') appears in the query, and in the second hypothesis of rule (R') itself. The first argument of the query is constant, so it takes only one value. The first argument of the second hypothesis is a variable z, which appears as the second argument of the first hypothesis, and thus takes $O(\texttt{\#edge.2})$ different values. Therefore, the number of invocations of rule (R') is $O(1 + \texttt{\#edge.2})$, which is $O(\texttt{\#edge.2})$.

The calculated complexities for rules (B'), (L'), and (R') are $O(\texttt{\#edge.2/1})$, $O(\texttt{\#path.2/1} \times \texttt{\#edge.2/1})$, and $O(\texttt{\#edge.2/1} \times \texttt{\#path.2/1} \times \texttt{\#edge.2})$, respectively. Therefore, the time complexity of the target query using left-recursion is $O(\texttt{\#path.2/1} \times \texttt{\#edge.2/1})$, and using right recursion is $O(\texttt{\#edge.2/1} \times \texttt{\#path.2/1} \times \texttt{\#edge.2})$.

3.3 Space complexity analysis

The asymptotic space complexity of tabled top-down evaluation is bounded by the space for table entries. Each table entry is keyed on an annotated predicate and values for the bound arguments. For an annotated predicate, the space it takes is the product of: (1) *number of table entries created*—the number of values that the bound arguments can take in subqueries of the annotated predicate, and (2) *size of each table entry*—the number of different values that the free arguments can take in the facts inferred for the annotated predicate. We give a method to calculate an upper bound for each factor. Summing the space used for all predicates gives the total space.

The number of table entries created for an annotated predicate p is calculated as follows. First, among all hypotheses of all rules and the given query, find those whose predicate is p. Then, for each such hypothesis, perform the second and third step of the method for computing the number of invocations in the previous subsection.

For example, for the left-recursive version of transitive closure and target query path(c,y)?, the number of table entries created for the predicate path_bf is $O(1)$ due to the given query and rule (L'), since the first hypothesis of (L') is a variant of its conclusion. For the right-recursive version and the same query, the number of table entries created for path_bf is $O(\texttt{\#edge.2})$ due to the query and rule (R').

The size of each table entry for each annotated predicate p is calculated as follows. For each rule r that defines p, we calculate the number of values that the free variables of the conclusion can take. Each of these free variables can take $O(\texttt{\#q.i})$ different values if it is the ith argument of a hypothesis of r whose predicate is q; if there are multiple such hypotheses, the minimum of these is taken. The product of the numbers of different values that the free variables of the conclusion can take in r is the size of each table

entry for facts inferred by r. The sum over all rules gives the final size of each table entry.

For example, consider the left-recursive version of transitive closure, and the target query. For the size of each table entry of path_bf, rule (B') incurs $O(\#edge.2)$ due to the first hypothesis, and rule (L') incurs $O(\#edge.2)$ due to the second hypothesis. Therefore, the total size of each table entry is $O(\#edge.2)$. The number of table entries created for path_bf is $O(1)$ as shown above. Therefore, the total space complexity is $O(1 \times \#edge.2)$. Using this analysis, the space complexity for the right-recursive version for the target query is $O(\#edge.2 \times (\#edge.2 + \#path.2))$.

Besides estimating memory usage, space complexity analysis can also help compare the actual running time for queries that have the same asymptotic time complexity. Creating a table entry is more expensive than adding a fact to a table entry in implementations such as XSB [25]. Therefore, the query that creates fewer table entries uses a constant factor less memory, and runs a constant factor faster.

For example, consider the left-recursive and right-recursive version of transitive closure. Given a query where both arguments are bound, the time complexities of both versions are the same. The left-recursive version contains rule (L'), for which the number of table entries is analyzed above. However, annotated rules for the right-recursive version contains the rule (R'').

```
path_bb(x,y) :- edge_bf(x,z), path_bb(z,y). (R'')
```

The second hypothesis of (R'') creates $O(\#edge.2)$ table entries, in contrast to $O(1)$ table entries created by (L'). Therefore, we conclude that the right-recursive version should run a constant factor slower. The experiments in Section 6 confirm this.

4. Demand-driven bottom-up evaluation

Bottom-up evaluation starts with given facts, infers new facts from conclusions of rules whose hypotheses match existing facts, and does so repeatedly until all facts are inferred. We use the bottom-up evaluation method of [21]. For best time complexity, this method decomposes any rule that has more than two hypotheses into a set of rules of two hypotheses; we decompose the hypotheses from left to right. We call this method *left-optimal bottom-up evaluation*, because the time complexity of evaluating a set of rules using this method is optimal for the fixed left-to-right ordering of the hypotheses in a rule.

- The time complexity incurred by each rule for this method is the *number of firings* of the rule—the number of combinations of facts that make all hypotheses true.
- The space complexity of this method consists of the space used by the inferred facts, and the space used by auxiliary maps as indices for constant time retrieval of relevant facts.

For the rest of the paper, *bottom-up evaluation* refers to left-optimal bottom-up evaluation.

Bottom-up evaluation infers all facts that can possibly be inferred without taking the given query into account, and thus may take asymptotically more time than necessary. To take the query into account, we perform *demand transformation*.

- Demand transformation transforms the given set of rules and query into a new set of rules and a fact, so that bottom-up evaluation using the new rules and fact, for any given set of facts, infers only useful facts for answering the query. It achieves this by mimicking top-down evaluation of the given query q so that for predicates in the given rules, only facts that would be inferred during tabled top-down evaluation of q are inferred in a bottom-up evaluation of the transformed rules.
- We also show that demand transformation can be obtained by simplifying the output of the well-known magic set transformation (MST). Annotations in MST are not necessary using bottom-up evaluation, because the indices corresponding to the annotations are generated automatically. Therefore, the output of our transformation is simpler.

For the rest of the paper, *demand-driven bottom-up evaluation* refers to performing bottom-up evaluation on the rules generated by demand transformation.

4.1 Demand transformation

To perform demand transformation, we first compute demand patterns as shown in Section 3.1. Then, for each demand pattern $\langle \mathtt{p}, \mathtt{s} \rangle$, and for each rule

$\mathtt{p(...)}$:- $\mathtt{h}_1$,..., $\mathtt{h}_n$.

the following rule is generated

$\mathtt{p(...)}$:- $\mathtt{d\_p\_s(a}_1\mathtt{,...,a}_k\mathtt{)}$, $\mathtt{h}_1$,..., $\mathtt{h}_n$.

where $\mathtt{a}_1$,...,$\mathtt{a}_k$ are the arguments of the conclusion bound by s. The new hypothesis is added to ensure that only facts that would be inferred in tabled top-down evaluation are inferred. Then, a fact and rules that define the facts of each predicate d_p_s are generated. For the given query, $\mathtt{p(a}_1\mathtt{,...,a}_k\mathtt{)?}$, the following fact is generated

$\mathtt{d\_p\_s(a}_{b1}\mathtt{,...,a}_{bl}\mathtt{)}$.

where $\mathtt{a}_{b1}$,...,$\mathtt{a}_{bl}$ are the constant arguments of the query, and s is the pattern string of the query. For each rule r generated, c :- $\mathtt{h}_0$,..., $\mathtt{h}_n$., and for each $\mathtt{h}_i$ whose predicate is an IDB predicate p, the following rule is generated

$\mathtt{d\_p\_s(a}_1\mathtt{,...,a}_k\mathtt{)}$:- $\mathtt{h}_0$,..., $\mathtt{h}_{i-1}$.

where $\mathtt{a}_1$,..., $\mathtt{a}_k$ are the bound arguments of $\mathtt{h}_i$, and s is the pattern string of $\mathtt{h}_i$.

For example, for rules (B) and (L), and target query path(c,y)?, the set of demand patterns is $\{\langle \mathtt{path}, \mathtt{'bf'} \rangle\}$. Demand transformation generates the following fact and rules.

```
d_path_bf(c).                                  (F)
path(x,y) :- d_path_bf(x), edge(x,y).          (Bd)
path(x,y) :- d_path_bf(x), path(x,z),          (Ld)
             edge(z,y).
d_path_bf(x) :- d_path_bf(x).                  (D)
```

Fact (F) corresponds to the given query. Rules (Bd) and (Ld) correspond to the demand pattern $\langle \mathtt{path}, \mathtt{'bf'} \rangle$. Rule (D) is for the second hypothesis of rule (Ld). Bottom-up evaluation using the generated rules has smaller time complexity, because in the given rule (L), the variable x could take an arbitrary value, whereas in rule (Ld), its value is restricted by the new hypothesis, d_path_bf, for which only one fact, (F), exists, so x can only be c.

Note that demand transformation does not necessarily reduce the asymptotic time complexity. Consider rules (B) and (L), and source query path(x,c)?, instead of target query. The set of demand patterns is $\{\langle \mathtt{path}, \mathtt{'fb'} \rangle, \langle \mathtt{path}, \mathtt{'ff'} \rangle\}$. Demand transformation generates the following fact and rules.

```
d_path_fb(c).
path(x,y) :- d_path_fb(y), edge(x,y).
path(x,y) :- d_path_fb(y),
             path(x,z), edge(z,y).
path(x,y) :- d_path_ff(), edge(x,y).
path(x,y) :- d_path_ff(),
             path(x,z), edge(z,y).
d_path_ff() :- d_path_fb(y).
d_path_ff() :- d_path_ff().
```

The time complexity of bottom-up evaluation using the generated rules is not better than the original rules if the underlying graph is connected, since no variable is restricted analogous to `x` in the previous example. For demand transformation to improve the complexity for target query as it did for source query, the left-recursive rule needs to be transformed into a right-recursive rule using recursion conversion [29].

4.2 Comparing with magic set transformation

Magic set transformation (MST) has the same goal as demand transformation. MST has three similar steps: binding annotation, generating rules and adding hypotheses for reflecting the demand of computation, and generating a fact for the demand by the query. A detailed description of the MST algorithm can be found in [30].

The disadvantage of MST, in contrast to demand transformation, is that it annotates the IDB predicates in the generated rules, and this may result in exponentially increased space complexity in program size, as shown below.

Consider the last example in the previous subsection. MST yields the following fact and rules.

```
d_path_fb(c).
path_fb(x,y) :- d_path_fb(y), edge(x,y).
path_fb(x,y) :- d_path_fb(y),
                path_fb(x,z), edge(z,y).
path_ff(x,y) :- d_path_ff(), edge(x,y).
path_ff(x,y) :- d_path_ff(),
                path_ff(x,z), edge(z,y).
d_path_ff() :- d_path_fb(y).
d_path_ff() :- d_path_ff().
```

The difference is the extra annotations in the annotated `path` predicates in the rules generated by MST. These rules may infer some same facts of `path` for two differently annotated predicates, `path_fb` and `path_ff`.

In general, annotating IDB hypotheses with their pattern strings is not necessary, because using bottom-up evaluation, indices for matching hypotheses are created automatically. Removing these annotations yields simpler rules, and reduces space taken by the same fact duplicated for multiple new predicates generated. The extra space from keeping the annotations is exponential in the number of arguments of differently annotated predicates.

Removing annotations of IDB predicates in the generated rules by MST yields the rules generated by demand transformation.

5. Relating tabled top-down and demand-driven bottom-up evaluations

5.1 Time complexity comparison

We establish the time complexity relationship between tabled top-down and demand-driven bottom-up evaluation. First, we show the relationship in the general case, then identify a subset of Datalog for which the two evaluations are equivalent, and finally show that adding early completion may improve tabled top-down evaluation.

Theorem 1 states that demand-driven bottom-up evaluation is faster than or equal to tabled top-down evaluation in time complexity.

Theorem 1. *Let P be a set of Datalog rules and a query. Let P' be the set of rules and fact after demand transformation of P. Let T_{td} be the asymptotic time complexity of tabled top-down evaluation of P, and T_{bu} be the asymptotic time complexity of bottom-up evaluation of P'. Then, $T_{bu} \leq T_{td}$.*

Proof. Let P_a be the set of rules and query after annotating P.

T_{td} is the sum of the complexities incurred by each rule in P_a. For each rule r in P_a of the form `p(...) :- body.`, there is a rule r' of the form `p(...) :- d(...), body.` in P', where `d(...)` is the new demand hypothesis. The complexity incurred by r for T_{td} is $i \times l$, where i is *the number of invocations* to r, and l is *the local complexity*, and l is the product of the sizes of the hypotheses. Since facts of `d` are obtained from all of the call sites to `p` with the same binding pattern as top-down evaluation, `#d` $= i$. For T_{bu}, the complexity incurred by a rule is the number of times the rule fires. Therefore, the complexity incurred by r' has an upper bound `#d`$\times l = i \times l$.

The only rules in P' that do not correspond to a rule in P_a are the rules that infer facts of the predicates added for demand. The additional complexity incurred for T_{bu} by each such rule is already dominated by a component of the complexity in T_{td}, because this complexity equals the number of invocations for the rule that the demand hypothesis would be added to, and the number of invocations is used as a factor in a summand of T_{td}.

Hence, $T_{bu} \leq T_{td}$. □

A rule is said to have no *singleton variables*, i.e. no *wildcards*, if it has only one hypothesis, or if each variable v that appears in a hypothesis also appears in another hypothesis or the conclusion.

We show that for Datalog rules with no more than two hypotheses, and no singleton variables, the time complexities of tabled top-down evaluation and demand-driven bottom-up evaluation are equal.

Lemma 2. *In bottom-up evaluation, if all variables in the hypotheses of a rule r are also in the conclusion of r, then the number of facts inferred using r equals the number of firings of r.*

Proof. The number of facts inferred using r cannot be larger than the number of firings of r, since a fact is inferred only in a firing of r.

Let f_1 and f_2 be two different firings of a rule r. There is at least one variable whose value is different between f_1 and f_2. Since all variables in the hypotheses also appear in the conclusion, the facts inferred from f_1 and f_2 must be different.

Therefore, the number of facts inferred using r equals the number of firings of r. □

Theorem 3. *Let P be a set of Datalog rules and a query, such that the rules have no singleton variables, and there are no more than two hypotheses per rule. Let P' be the set of rules and fact after demand transformation of P. Let T_{td} be the asymptotic time complexity of tabled top-down evaluation of P, and T_{bu} be the asymptotic time complexity of bottom-up evaluation of P'. Then, $T_{bu} = T_{td}$.*

Proof. Let P_a be the set of rules after annotating the rules in P. Each rule r in P_a is of one of two forms:

(i) r has one hypothesis, so has the form `c :- h.` In P', there is a rule r' corresponding to r, and is of the form `c :- d, h.`, where `d` is the new demand hypothesis. The complexity incurred by r' to T_{bu} and by r to T_{td} are both dominated by the size of the predicate of `h`, since `h` contains all variables in `d`.

(ii) r has two hypotheses, so has the form `c :-` $\mathtt{h}_1$`,` $\mathtt{h}_2$`.` In P', there is a rule r' corresponding to r, and is of the form `c :- d,` $\mathtt{h}_1$`,` $\mathtt{h}_2$`.`, where `d` is the demand hypothesis added. As before, the complexity incurred by r to T_{td}, denoted $T_{td}(r)$, equals the product of the sizes of the predicates `d`, $\mathtt{h}_1$, and $\mathtt{h}_2$. However, bottom-up computation can decompose the rules to possibly improve performance. In this case, it would obtain the following two rules: `new :- d,` $\mathtt{h}_1$`.` and `c :- new,` $\mathtt{h}_2$. The complexity of the first rule is less than $T_{td}(r)$. Since there are no singleton variables, the variables of `d` and $\mathtt{h}_1$ must appear in `new`. Then, by Lemma 2, the

size of the predicate of `new` equals the running time of the rule that generates it, and hence the complexity incurred by the second rule obtained from r' equals $T_{td}(r)$.

Therefore, for each complexity summand incurred by rules in P_a for T_{td}, there is a rule in P' that incurs the same complexity summand for T_{bu}. Combining this with Theorem 1, which states that $T_{bu} \leq T_{td}$, we obtain $T_{bu} = T_{td}$. □

Early completion is an optimization for tabled top-down evaluation. It stops backtracking for queries with all arguments bound immediately after they are proven to be true. Theorem 4 states that adding early completion to tabled top-down evaluation may make it asymptotically faster than demand-driven bottom-up evaluation.

Theorem 4. *The time complexity of tabled top-down evaluation with early completion can be smaller than demand-driven bottom-up evaluation.*

Proof. With early completion, tabled top-down evaluation stops backtracking when it proves that a subquery with all arguments bound is true, whereas bottom-up evaluation always exhausts all possible ways of proving facts. Therefore, with early completion, the time complexity of tabled top-down evaluation can be smaller than demand-driven bottom-up evaluation. □

5.2 Space complexity comparison

We establish the space complexity relationship between tabled top-down and demand-driven bottom-up evaluation. We first show the relationship in the general case. We then show that if demand-driven bottom-up evaluation has better complexity, then its space complexity must be worse.

Theorem 6 states that tabled top-down evaluation does not use asymptotically more space than demand-driven bottom-up evaluation. We prove it by showing the components of space used for demand-driven bottom-up evaluation and their correspondence with the space usage in tabled top-down evaluation.

Lemma 5. *Let P be a set of Datalog rules and a query. Let P' be the set of rules and fact after demand transformation of P. For each predicate p in P, let $BU(p)$ be the set of facts of p inferred using bottom-up evaluation of P', and let $TD(p)$ be the set of facts of p inferred during tabled top-down evaluation of P. Then, $BU(p) = TD(p)$.*

Proof. We showed in Theorem 1 that the bound argument values of subqueries for which each rule will be invoked in tabled top-down computation is a fact for the demand hypothesis added in bottom-up computation. Therefore, for each predicate p, the same facts will be inferred by each rule that defines p using either method. Hence, $BU(p) = TD(p)$. □

Theorem 6. *Let P be a set of Datalog rules and a query. Let P' be the set of rules and fact after demand transformation of P. Let S_{td} be the asymptotic space complexity of tabled top-down evaluation of P, and S_{bu} be the asymptotic space complexity of bottom-up evaluation of P'. Then, $S_{td} \leq S_{bu}$.*

Proof. S_{bu} consists of the sums of each of the following items: (i) the number of facts of each predicate defined by rules, (ii) the number of facts of each demand predicate, (iii) the number of facts of each predicate defined for decomposing rules into rules with at most two hypotheses, and (iv) the size of the auxiliary maps maintained for fact retrieval. S_{td} consists only of the sum of the sizes of table entries for each predicate defined by rules.

For a predicate p, by Lemma 5, the set of facts of p inferred by either evaluation method is the same. Each fact of p is stored only once in the bottom-up method, but they can be stored in 2^k auxiliary maps, where k is the number of arguments of p. For tabled top-down evaluation, each fact of p may be stored in at most 2^k tables, where the number of table entries correspond exactly to the auxiliary maps. Therefore, $S_{td} \leq S_{bu}$. □

Theorem 7 states that improvement in time complexity for demand-driven bottom-up evaluation is only possible by using more space. We prove it by using the fact that such improvement is only possible by using more space in decomposed rules.

Theorem 7. *Let P be a set of Datalog rules and a query. Let P' be the set of rules and fact after demand transformation of P. Let T_{td} and S_{td} be the asymptotic time and space complexity of tabled top-down evaluation of P, and T_{bu} and S_{bu} be the asymptotic time and space complexity of bottom-up evaluation of P'. Then, if $T_{bu} < T_{td}$, then $S_{td} < S_{bu}$.*

Proof. If $T_{bu} < T_{td}$, then in the bottom-up evaluation of P, there is a rule which is decomposed into multiple rules for bottom-up evaluation. This implies that the third component of S_{bu} shown in Theorem 6 is nonzero. Since S_{td} only consists of the first and fourth item of S_{bu}, $S_{td} < S_{bu}$. □

6. Experiments

We support our complexity analyses and comparisons by experiments. For tabled top-down evaluation, we use XSB [33]. For bottom-up evaluation, we use the implementation method of [21] to generate Python code from the rules.

We examined all benchmarks in OpenRuleBench [18]. All benchmark rules can be classified as pure joins (no recursion), transitive closure, and same generation (whether two nodes are in the same generation in trees). We show experimental results for three benchmarks, one for each class.

We instantiate the complexity parameters in predicted complexities with their values computed from the data. We use *space units* to mean number of unique table inserts for tabled top-down, and the number of facts inferred plus the number of elements in auxiliary maps for demand-driven bottom-up evaluation. We use *returns* to mean the number of total facts returned from rules for tabled top-down evaluation, and *firings* to mean the number of firings for demand-driven bottom-up evaluation.

In all three benchmarks, the predicates have two arguments. For experiments, we fix `#p` and `#p.1/2` for each input predicate `p` to generate a set of data such that the size of each predicate is maximal, i.e., the worst-case behavior is exhibited. Then, we increase `#p` and `#p.1/2` to generate the next set of data, and repeat.

For pure joins, we show results for the benchmark *Join1*, which contains 4 rules that join 5 predicates, with a query with all arguments free. Figure 2 shows that returns for tabled top-down evaluation and firings for demand-driven bottom-up evaluation are linear in predicated time units. It also shows that the space units for both is linear in predicted space units. These confirm our analyses. The time difference between tabled top-down and demand-driven bottom-up evaluations arise from the rules that infer demand. The space difference between them arise from demand predicates and auxiliary maps.

For transitive closure, we analyze the time complexity, and by using the space complexity analysis for tabled top-down evaluation, give a comparison of actual running times when the asymptotic time complexity is the same. We showed in Section 3 that the left- and right-recursive versions of transitive closure have the same asymptotic time and space complexities for a query with both arguments bound, but the right-recursive version creates asymptotically more table entries. Therefore, the right-recursive version will run slower by a constant factor, and use a constant factor more space.

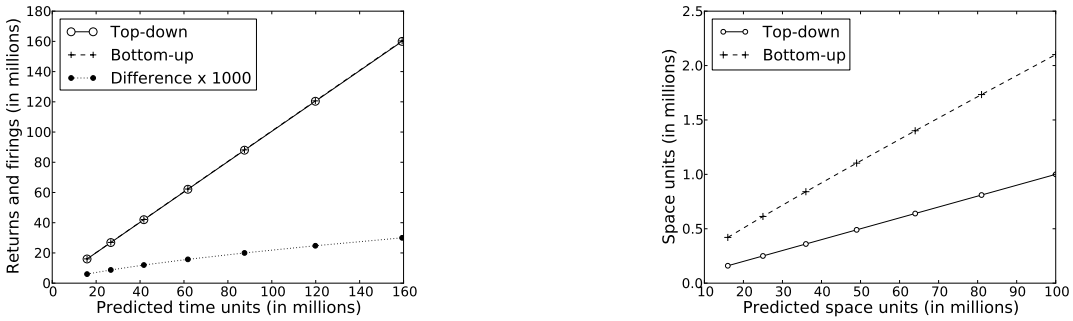

Figure 2. Returns for tabled top-down evaluation, firings for bottom-up evaluation, and space units for both, for benchmark *Join1*. The difference between returns and firings has been multiplied by 1000 for illustration.

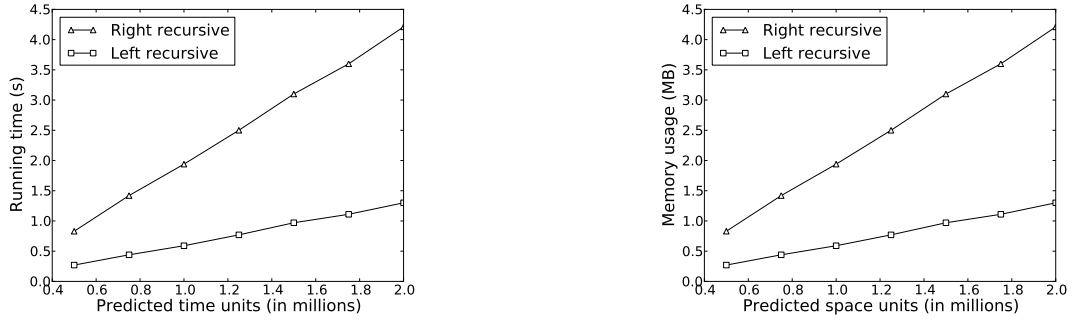

Figure 3. Running time and memory usage of transitive closure in XSB for a query with both arguments bound.

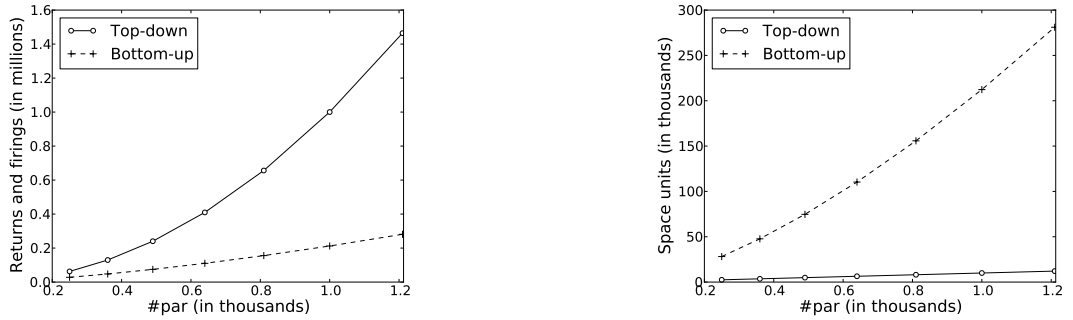

Figure 4. Returns for tabled top-down evaluation, firings for bottom-up evaluation, and space units for both, for the same generation benchmark.

Figure 3 confirms that their complexities are the same, since the actual time and space are linear in the predictions. It also confirms that the right-recursive version uses a constant factor more time and space.

The same generation benchmark contains a rule with three hypotheses: `sg(x,y) :- par(x,z1), sg(z1,z2), par(y,z2).` We use the query `sg(c,y)?`, and show the time and space tradeoff. Bottom-up evaluation eliminates variable z_1 after decomposing the rules into rules with two hypotheses, and has better time complexity than tabled top-down evaluation. Therefore, tabled top-down evaluation has better space complexity. Figure 4 confirms our analysis: the returns of tabled top-down evaluation increases faster asymptotically, and the space units of tabled top-down evaluation increases slower asymptotically.

7. Related work and conclusion

Datalog has been extensively studied [1, 7]. Tabled top-down evaluation was introduced in [28], and an implementation of it is described in [9]. Optimal bottom-up evaluation, on which our left-optimal bottom-up evaluation is based, is described in [21].

For tabled top-down evaluation of Datalog, the only known bound on the time complexity is $O(k^v)$, where k is the number of constants in the input data, and v is the maximum number of variables in a rule [33], and there is no complexity analysis studied for space. Our method calculates worst-case time complexity much more precisely, and is the first to calculate worst-case space complexity and calculates it precisely.

For bottom-up evaluation, time and space complexities have been analyzed before, using prefix-firing by Ganzinger et al. [14]

and optimal bottom-up evaluation by Liu et al. [21]. Bottom-up evaluation was used to mimic top-down evaluation after program transformations, mostly notably magic set transformation [3]. Our demand transformation is simpler and produces simpler rules that have the same time and space in terms of data complexity and exponentially smaller space in terms of program complexity.

The relationship between top-down and bottom-up evaluation has been studied [24]. Ullman [30] shows that bottom-up evaluation after magic set transformation has better than or equal time complexity with a breadth-first top-down strategy called QRGT without tabling. Ramakrishnan et al. [23] describe magic set transformation with tail recursion optimization that is better than or equal to than top-down evaluation with tail recursion optimization. Bry [4] shows that top-down evaluation with variant tabling and bottom-up evaluation after magic set transformation can be implemented in a unified framework, and that they infer the same facts for the given predicates, but does not study time and space complexities. Our work is the first to establish precise relationships between tabled top-down evaluation and demand-driven bottom-up evaluation in terms of precise time and space complexities.

Additionally, we have implemented our method and confirmed our analysis results through experiments on well-studied benchmarks.

The complexity results can be used for optimizations by comparing the complexity formulas of different rules with the same semantics. However, comparison of complexity formulas may be difficult in general, in which case estimations of size parameters [20] can be used to help. Future work includes study of powerful methods for automatically simplifying complexity formulas, for estimating values of size parameters, and for using our method for optimizations.

Acknowledgments. We thank David Warren and Michael Kifer for many helpful discussions on the implementation of XSB, and on logic programming in general.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. of the 1986 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD)*, pages 16–52, 1986.

[3] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *J. Logic Programming*, 10(1/2/3&4):255–299, 1991.

[4] Francois Bry. Query evaluation in deductive databases: Bottom-up and top-down reconciled. *Data Knowledge Engineering*, 5:289–312, 1990.

[5] Stefano Ceri, Georg Gottlob, and Luigi Lavazza. Translation and optimization of logic queries: The algebraic approach. In *Proc. of the 12th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 395–402, 1986.

[6] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowledge and Data Engineering*, 1(1):146–166, 1989.

[7] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer, 1990.

[8] Stefano Ceri and Letizia Tanca. Optimization of systems of algebraic equations for evaluating datalog queries. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *Proc. of the 13th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 31–41. Morgan Kaufmann, 1987.

[9] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[10] Anderson Faustino da Silva and Vítor Santos Costa. The design of the YAP compiler: An optimizing compiler for logic programming languages. *J. of Universal Computer Science*, 12(7):764–787, 2006.

[11] Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. Type inference for datalog and its application to query optimisation. In *Proc. of the 27th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*, pages 291–300, 2008.

[12] John DeTreville. Binder, a logic-based security language. In *Proc. of the 2002 IEEE Symp. on Security and Privacy (S&P)*, pages 105–113, 2002.

[13] Juliana Freire, Terrance Swift, and David S. Warren. Beyond depth-first strategies: Improving tabled logic programs through alternative scheduling. *J. of Functional and Logic Programming*, 1998(3), 1998.

[14] Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proc. of the 1st Intl. Joint Conf. on Automated Reasoning (IJCAR)*, pages 514–528, 2001.

[15] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *CodeQuest:* Scalable source code queries with Datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming (ECOOP)*, pages 2–27, 2006.

[16] Lawrence J. Henschen and Shamim A. Naqvi. On compiling queries in recursive first-order databases. *J. ACM*, 31(1):47–85, 1984.

[17] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proc. of the 5th Intl. Symp. on Practical Aspects of Declarative Languages (PADL)*, pages 58–73, 2003.

[18] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. OpenRuleBench: An analysis of the performance of rule engines. In *Proc. of the 18th Intl. Conf. on World Wide Web (WWW)*, pages 601–610, 2009.

[19] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. OpenRuleBench: Detailed report. Technical report, Department of Computer Science, Stony Brook University, 2009. Available at http://semwebcentral.org/docman/view.php/158/69/report.pdf.

[20] Senlin Liang and Michael Kifer. Deriving predicate statistics in Datalog. In *Proc. of the 12th Intl. ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, 2010.

[21] Yanhong A. Liu and Scott D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Trans. Programming Languages and Systems*, 31(6), 2009.

[22] Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Y. Vardi. Logical query optimization by proof-tree transformation. *J. of Computer and System Sciences*, 47(1):222 – 248, 1993.

[23] Raghu Ramakrishnan and S. Sudarshan. Top-down versus bottom-up revisited. In *Proc. of of the 1991 Intl. Symp. on Logic Programming (ISLP)*, pages 321–336, 1991.

[24] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *J. Logic Programming*, 23(2):125–149, 1995.

[25] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as a deductive database. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD)*, page 512, 1994.

[26] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *Proc. of the 33rd Intl. Conf. on Very Large Data Bases (VLDB)*, pages 1033–1044, 2007.

[27] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, August 1997.

[28] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In *Proc. of the 3rd Intl. Conf. on Logic Programming (ICLP)*, pages 84–98, 1986.

[29] K. Tuncay Tekle, Katia Hristova, and Yanhong A. Liu. Generating specialized rules and programs for demand-driven analysis. In *Proc. of the 12th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST)*, pages 346–361, 2008.

[30] Jeffrey D. Ullman. Bottom-up beats top-down for Datalog. In *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)*, pages 140–149, 1989.

[31] Laurent Vieille. A database-complete proof procedure based on SLD-resolution. In *Proc. of the 4th International Conference on Logic Programming (ICLP)*, pages 74–103, 1987.

[32] Laurent Vieille. From QSQ towards QoSaQ: Global optimization of recursive queries. In *Proc. from the 2nd Intl. Conf. on Expert Database Systems (EDS)*, pages 743–778, 1988.

[33] David S. Warren. Programming in tabled Prolog. Available at http://www.cs.sunysb.edu/∼ warren/xsbbook/, 1999.

[34] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proc. of the 3rd Third Asian Symp. on Programming Languages and Systems (APLAS)*, pages 97–118, 2005.