More Efficient Datalog Queries: Subsumptive Tabling Beats Magic Sets*

K. Tuncay Tekle
LogicBlox, Inc.
Atlanta, GA
tuncay@cs.sunysb.edu

Yanhong A. Liu
Department of Computer Science
State University of New York at Stony Brook
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. The dominant strategies to improve the performance of answering queries are reusing answers to subqueries for top-down methods, and transforming rules based on demand from the query, such as the well-known magic sets transformation, for bottom-up methods. However, the performance of these strategies vary drastically, and the most effective method remained unknown.

This paper describes precise time and space complexity analysis for efficient implementation of Datalog queries using subsumptive tabling, a top-down evaluation method with more reuse of answers than the dominant tabling strategy, and shows that subsumptive tabling beats bottom-up evaluation of rules after magic sets transformation in both time and space complexities. It also describes subsumptive demand transformation, a novel method for transforming the rules so that bottom-up evaluation of the transformed rules mimics subsumptive tabling; we show that the time complexity of bottom-up evaluation after this transformation is equal to the the time complexity of top-down evaluation with subsumptive tabling. The paper further describes subsumption optimization, an optimization to increase the use of subsumption in subsumptive methods, and shows its application in the derivation of a well-known demand-driven pointer analysis algorithm. We support our analyses and comparisons through experiments with applications in ontology queries and program analysis.

*This work was supported in part by NSF under grants CCF-0964196 and CCF-0613913 and by ONR under grant N000140910651. This work was carried out in part while the first author was a Ph.D. student at the State University of New York at Stony Brook.

SIGMOD'11, June 12–16, 2011, Athens, Greece.

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 [5] is a logic language for deductive databases [1], program analysis [30], security [9], networking [17], and many other applications [14, 10, 24, 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. The dominant strategies for efficient evaluations are top-down evaluation with *variant tabling* [25] to memoize and reuse answers to subqueries, and bottom-up evaluation with the *magic set transformation* (MST) [3] so that evaluation of the transformed rules are driven by demand from the query.

For a subquery encountered during top-down evaluation with variant tabling, only answers from identical subqueries are reused. However, there may exist another subquery already encountered that *subsumes* the current subquery, i.e., whose answers are guaranteed to contain all answers to the current subquery. Using this, top-down evaluation can be coupled with a tabling strategy, called *subsumptive tabling* [21], that performs more reuse of previously inferred answers.

The performance of Datalog engines is difficult to understand due to the multitude of factors involved. For example, the performance of different tabling strategies vary drastically [21], and bottom-up evaluation after MST may be much slower than the bottom-up evaluation of the original rules [23]. Choosing the best evaluation method for a given

set of rules requires precise characterization of the time and space complexities of each evaluation method.

Despite extensive research on optimization of Datalog rules [7, 22, 18, 13], given a set of rules and a query, no transformations that yield better time complexity than MST are known. *Demand transformation* [27] is similar to MST, with better space complexity in program size, but the same time complexity. There exists no transformation such that the bottom-up evaluation of transformed rules achieves the performance of subsumptive tabling.

This work describes precise time and space complexity analysis for efficiently answering Datalog queries using subsumptive tabling, and precise relationships between top-down evaluations with variant and subsumptive tabling, and their relationship to bottom-up evaluation after MST. We give complexity analyses for top-down evaluation with subsumptive tabling by determining the binding patterns of arguments for queries, and the subqueries that are guaranteed to reuse answers from subsuming subqueries, and then extending the analysis for variant tabling [27] to subsumptive tabling. We show that top-down evaluation using subsumptive tabling is equal to or better than using variant tabling in both time and space complexities. Using this result and the relationship between top-down evaluation with variant tabling and bottom-up evaluation after MST as shown in [27], we show that subsumptive tabling is equal to or better than MST in both time and space complexities. Then, we characterize a class of Datalog rules for which subsumptive tabling is guaranteed to be better than variant tabling in both time and space complexities.

Additionally, we describe a transformation, called *subsumptive demand transformation* (SDT), such that bottom-up evaluation of the rules produced by SDT achieves the performance of subsumptive tabling. We modify bottom-up evaluation slightly, and couple it with SDT to obtain *subsumptive bottom-up evaluation*. We show that the time and space complexities of subsumptive bottom-up evaluation and subsumptive tabling are equal for rules with no more than two hypotheses each and no singleton variables; and that the time complexity of subsumptive bottom-up evaluation may be better than subsumptive tabling for other rules. We also show that using SDT is equal to or better than using MST. We show that for rules for which subsumptive tabling outperforms variant tabling, SDT outperforms MST.

Building on our analyses, we devise a transformation, called *subsumption optimization*, for making sure that a query that subsumes another in subsumptive tabling is performed first when it is better to do so in time complexity. Using this method, we show how to systematically derive Heintze and Tardieu's demand-driven pointer analysis [11] from the definition of Andersen's pointer analysis.

We show experimental results on an illustrative set of Datalog rules, rules for Andersen's pointer analysis [2], and rules for ontology queries from OpenRuleBench [15]. We first confirm our complexity analyses, and then confirm when subsumptive tabling and SDT are necessary for efficient evaluation of queries.

The rest of the paper is organized as follows. Section 2 defines Datalog, evaluation methods, and terminologies. Section 3 presents complexity analysis of subsumptive tabling, and the relationships with variant tabling and magic sets. Section 4 describes subsumptive demand transformation, and its relationship with the other methods. Section 5 presents subsumption optimization and its application to pointer analysis. Section 6 presents experimental results. Section 7 discusses related work and concludes.

2. DATALOG AND EVALUATION METHODS

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.

Tabling for top-down evaluation. Top-down evaluation starts with a given 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 generates repeatedly until the subqueries match given facts. This may lead to repeated subqueries or, when recursive rules exist, infinite recursion. To address this problem, *tabling* memoizes answers to subqueries, and reuses them when possible.

Variant tabling [6] 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.

Subsumptive tabling [21] reuses more answers by considering previous subqueries that *subsume* a new query, not only previous queries that are variants of the new query. A subquery q_1 *subsumes* subquery q_2 if there is a substitution θ of variables such that $\theta(q_1) = q_2$.

Precise complexity analysis for variant tabling has been studied recently [27]. We describe precise complexity analysis for subsumptive tabling in Section 3.

Demand-driven transformations for 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. In this work, by *bottom-up evaluation*, we refer to the optimal bottom-up evaluation method of Liu and Stoller [16] that can be systematically analyzed for time and space complexities.

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, a source-level transformation, like the well-known magic set transformation (MST) [3], is performed. *Demand transformation* [27] is such a transformation that results in rules with the same time complexity and better space complexity in program size by improving the annotations used in MST. It transforms a set of rules and a query into a new set of rules, such that the set of facts that can be inferred from the new set of rules contains only facts that would be inferred during a top-down evaluation of the original rules with variant tabling. We call this transformation *variant demand transformation* in this paper.

There exists no transformation analogous to subsumptive tabling for bottom-up evaluation. We develop the first such transformation, called *subsumptive demand transformation*, in Section 4.

Terminology. We refer to the different evaluation methods described above as follows:

- *v-topdown*: Top-down evaluation with variant tabling
- *s-topdown*: Top-down evaluation with subsumptive tabling
- *v-bottomup*: Bottom-up evaluation after variant demand transformation
- *s-bottomup*: Bottom-up evaluation after subsumptive demand transformation

The asymptotic time complexities of the above methods are denoted $T_{v-topdn}$, $T_{s-topdn}$, $T_{v-botup}$, $T_{s-botup}$. Similarly, asymptotic space complexities are denoted with S and the corresponding subscripts. For space complexities, we do not consider the stack space used by top-down evaluation. Bottom-up evaluation does not use stack space. For both time and space, the denoted complexities are in data size. We assume that program size, including the number of rules, the number of hypotheses in rules, and the number of arguments of predicates, is constant.

A singleton variable in a Datalog rule is a variable that occurs in one hypothesis and not in other hypotheses or the conclusion. Any set of Datalog rules can be transformed easily into rules that have at most two hypotheses and no singleton variables. We say that rules in this form are in *minimal form*.

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

In examples, we use letters from `p` to `z` for variables, `c` for constants, and `a` for either constant or variable arguments of predicates.

For complexity calculation, we use the following notations.

- `#p`: the number of facts of predicate `p` (given or inferred), called the *size of* `p`.
- `#p.`$\mathtt{i}_1$`,...,`$\mathtt{i}_n$`/`$\mathtt{j}_1$`,...,`$\mathtt{j}_m$: the maximum number of combinations of values for the $\mathtt{i}_1$,...,$\mathtt{i}_n$th arguments of the facts of predicate `p`, given a fixed combination of values for the $\mathtt{j}_1$,...,$\mathtt{j}_m$th arguments.
- `#p.i`: the *actual* number of values of the ith argument of predicate `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 example. We define a predicate of being *related* as a running example. A person x is related to a person y if x is in an immediate family with y, or if there is a person u in an immediate family with another person v, u is related to x, and v is related to y. This relation can be defined in Datalog using the following two rules:

```
rel(x,y) :- imm(x,y).                        (1)
rel(x,y) :- imm(u,v), rel(u,x), rel(v,y). (2)
```

For example, given these two rules and a query `rel(x,y)?`, a subquery `rel(c,x)?` for some constant `c` will be generated from the second hypothesis of the second rule. In variant tabling, this subquery will be used to generate more subqueries. In subsumptive tabling, since the given query subsumes this subquery, the answers to the subquery will be looked up in the table entry for the given query.

3. COMPLEXITY ANALYSIS OF SUBSUMPTIVE TABLING FOR TOP-DOWN EVALUATION

For analyzing the time and space complexities of top-down evaluation with subsumptive tabling, we make the following assumptions:

- *Depth-first scheduling* is used. This selects the next subqueries to evaluate in a depth-first manner. For v-topdown, the selection of the scheduling strategy does not change the complexities, since each distinct subquery is guaranteed to be processed, regardless of when. But, for s-topdown, the order of evaluation of subqueries may change whether a subquery is processed or not, since a subsuming one may have been encountered before in an order, and may not have been encountered in another. We show complexity analyses for depth-first scheduling, and then describe the changes necessary for other dominant scheduling strategies.
- *No early completion* is used. This uses all relevant rules to infer answers to a subquery the first time it is evaluated, even when it is a subquery whose arguments are all bound and has been evaluated to be true.
- All IDB predicates are tabled. This allows for the best asymptotic time complexity.
- All predicates are perfectly indexed. So, it takes constant time to retrieve a fact of a predicate given fixed values for some of its arguments. Not using perfect indices would increase asymptotic complexity by the cost of retrieving facts in the absence of perfect indices. Mainstream implementations of tabling, such as XSB [6] and YAP [8], support perfect indexing.

In this section, we show the calculation of time and space complexities for s-topdown, that it is equal to or better than v-topdown, and that it is equal to or better than MST for rules in a certain minimal form, and we identify a class of rules for which s-topdown is guaranteed to outperform v-topdown and MST.

3.1 Subsumptive binding annotation and complexity analyses

The time complexity of s-topdown is the sum of the number of facts that match the hypotheses in the body of each rule for each subquery that is not looked up in the table. The space complexity is the number of facts stored in the table entries. These are impossible to determine statically, since it is not possible to determine whether a subquery will be looked up or evaluated.

We give a method for obtaining an upper bound on the complexities that is as close as possible to the actual complexities. For easier and more precise calculation of the complexities, we first generate a query and rules annotated with the patterns of argument bindings based on the given query, but whose evaluation using s-topdown is otherwise the same as the given query and rules. Then, we calculate the complexity of evaluating the annotated query and rules.

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 *subsumptive demand patterns*, and then generate an annotated rule for each pattern determined. This method is a generalization of the binding annotation method shown in [27] that is used for v-topdown.

Subsumptive demand patterns. For each subquery, we determine whether it is *guaranteed to be evaluated* during s-topdown, where a subquery is guaranteed to be evaluated if it will be evaluated during s-topdown regardless of what facts are given or inferred. Given a set of rules and a query, each subquery $\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ encountered during s-topdown yields an *s-demand pattern* $\langle \texttt{p}, n, r, g, s\rangle$, where the subquery is the nth hypothesis of rule r, s is a string, called the *pattern string*, of length k whose ith character is '`b`' if $\texttt{a}_i$ is bound, and '`f`' otherwise, and g is a boolean value indicating whether this subquery is guaranteed to be evaluated. 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`'.

An s-demand pattern d is said to subsume another s-demand pattern d_2 if d is guaranteed to be evaluated and any subquery with the pattern d subsumes subqueries with the pattern d_2. Formally, an s-demand pattern $\langle \texttt{p}, n, r, g, s\rangle$ subsumes an s-demand pattern $\langle \texttt{p}_2, n_2, r_2, g_2, s_2\rangle$ if g is `true`, and either (i) s contains no '`b`'s, or (ii) for each j such that the jth character of s is '`b`', the jth character of s_2 is '`b`', and the hypotheses of r to the left of its nth hypothesis is a subset of the hypotheses of r_2 to the left of its n_2th hypothesis.

S-demand patterns are computed iteratively as follows until no new s-demand patterns can be added. The s-demand pattern of the given query $\texttt{p(a}_1\texttt{,...,a}_k\texttt{)}$ is $\langle \texttt{p}, 1, \_, \texttt{true}, s\rangle$, where 1 and _ indicate that the given query can be regarded as the first (and only) hypothesis of a non-existing rule, and the ith character of s is '`b`' if $\texttt{a}_i$ is a constant, and '`f`' otherwise. For each computed s-demand pattern $\langle \texttt{p}, n, r, g, s\rangle$, for each rule r_2 that defines `p`, and for each jth hypothesis h of r_2 whose predicate is an IDB predicate, say, `q`, add an s-demand pattern $\langle \texttt{q}, j, r_2, g_2, s_2\rangle$, if this s-demand pattern is not subsumed by an s-demand pattern already computed, where g_2 is `true` iff (i) g is `true`, (ii) j is 1, and (iii) the ith character of s_2 is '`b`' if the ith argument of h is a constant, 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.

After s-demand patterns are computed, we take the projection of the first and last element of each pattern to obtain a set of predicate-annotation pairs, and use s-demand patterns to refer to only these pairs.

Annotation. For each s-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 s-demand pattern $\langle \texttt{p}, \texttt{s}\rangle$, and each rule of the form

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

we generate the rule

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

where for each $1 \le k \le n$, the ith character of $\texttt{s}_k$ is '`b`' if the ith argument of $\texttt{h}_k$ is a constant, 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 the rules in the running example and the query `rel(x,y)?`, we show the difference between the annotated rules for v-topdown and s-topdown. For s-topdown, the computed set of s-demand patterns is $\{\langle \texttt{rel}, \text{`}\texttt{ff}\text{'}\rangle\}$, since the given query with both arguments free subsumes all other subsequent subqueries for `rel`, and hence the query `rel_ff(x,y)?` and the following two annotated rules are generated.

```
rel_ff(x,y) :- imm(x,y).                  (1a)
rel_ff(x,y) :- imm(u,v), rel_bf(u,x),     (2a)
               rel_bf(v,y).
```

For v-topdown, since subsumption is not used, the set of demand patterns would be $\{\langle \texttt{rel}, \text{`}\texttt{ff}\text{'}\rangle, \langle \texttt{rel}, \text{`}\texttt{bf}\text{'}\rangle, \langle \texttt{rel}, \text{`}\texttt{bb}\text{'}\rangle\}$, and annotation results in the same annotated query, the two annotated rules above, plus the four annotated rules below:

```
rel_bf(x,y) :- imm(x,y).                  (1b)
rel_bf(x,y) :- imm(u,v), rel_bb(u,x),     (2b)
               rel_bf(v,y).
rel_bb(x,y) :- imm(x,y).                  (1c)
rel_bb(x,y) :- imm(u,v), rel_bb(u,x),     (2c)
               rel_bb(v,y).
```

To the best of our knowledge, subsumptive binding annotation is the first annotation method that considers subsuming queries, and is distinct from previous methods used for v-topdown [27] and predicate splitting [28].

Other scheduling strategies. For local scheduling, batched scheduling, or other scheduling strategies, the method for obtaining the demand patterns need to be modified. The idea is that for each scheduling strategy, one needs to determine a heuristic that identifies subqueries guaranteed to be evaluated. For example, for local scheduling, which retrieves all answers from a subquery before continuing to other hypotheses, a subquery is guaranteed to be evaluated if it is the first hypothesis of a rule whose conclusion is guaranteed to be evaluated. For batched scheduling, which retrieves one answer from a subquery and then continues to other hypothesis, only the given query and the first

hypothesis of the first rule that defines the given query is guaranteed to be evaluated.

Using these heuristics, the demand pattern identification method can be modified to obtain the relevant demand patterns, and the same annotation method can be used afterwards.

Time and space complexity analyses. After subsumptive binding annotation, we use the method in [27] on the annotated rules for computing time and space complexities.

The time complexity is the sum of asymptotic complexities incurred by all annotated rules. 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.

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 '$\texttt{f}$' in $\texttt{s}$, and $\texttt{b}_i$ is the index of the ith '$\texttt{b}$' in $\texttt{s}$.

The number of invocations of a rule is the sum of all values possibly taken by its bound arguments in all possible subqueries. This is calculated by considering the hypotheses in all rules that have the same annotation as the conclusion of the rule being analyzed. For each hypothesis in each rule whose annotation is the same as the conclusion of the analyzed rule, we determine how many values the bound arguments take by finding out which argument of which hypotheses bind those arguments.

For the running example, the local complexity of `(2a)` is $O(\texttt{\#imm} \times \texttt{\#rel.2/1} \times \texttt{\#rel.2/1})$, and the number of invocations is $O(1)$ because there are no bound arguments, based on the annotation of the conclusion being '$\texttt{ff}$'. Therefore, the time complexity incurred by `(2a)` is $O(\texttt{\#imm} \times (\texttt{\#rel.2/1})^2)$.

The space complexity of s-topdown is bound by the number of facts stored. This is the sum, over all s-demand patterns, of the product of (i) the number of distinct table entries for an annotation and (ii) the number of facts for each table entry.

The number of distinct table entries is the number of values that the bound arguments in the conclusion take for an annotated predicate. It can be calculated using the method for calculating the number of invocations in time complexity, where the annotated predicate being analyzed for space complexity is considered as the predicate of the conclusion of the rule being analyzed in the method for calculating the number of invocations.

The number of facts for each table entry is the number of values that the free arguments in the conclusion take for rules defining that annotated predicate. Each free argument in the conclusion is bound by one or more hypotheses, and the number of values it can take is the minimum of the number of values that argument can take in the hypotheses it appears in. The number of facts is calculated as product of the number of values that each free argument in the conclusion can take.

For the running example, the only s-demand pattern is $\langle\texttt{rel},\text{'}\texttt{ff}\text{'}\rangle$, therefore there is $O(1)$ table entries, and the number of facts for that table entry is $O(\texttt{\#rel})$. Hence, the space complexity is $O(\texttt{\#rel})$.

3.2 Subsumptive beats variant and magic sets

We show that s-topdown always beats v-topdown, and therefore beats the magic set transformation (MST) for a form of Datalog rules, which all Datalog rules can be reduced to.

The following lemma shows that the check for subsuming used in s-topdown is more expensive than the check for being variant in v-topdown.

Lemma 1. *Let q be an IDB subquery of k arguments. The first time q is encountered in top-down evaluation, the time complexity of table lookup for q is $O(k)$ in variant tabling, and $O(2^k)$ in subsumptive tabling.*

Proof. In variant tabling, the evaluation looks up whether there is a previous table entry with the same label up to variable renaming, and create it if not. This lookup can be trivially done in $O(k)$ time. Table entry creation can also be done in $O(k)$ time.

In subsumptive tabling, the evaluation needs to look up for every possible subsuming query, whether there is a table with that label. If the query has b bound arguments, there are 2^b queries that subsume q, and b is $O(k)$, therefore table lookup takes $O(2^k)$ time. Table entry creation can be done in $O(k)$ time. Therefore, the total time complexity is $O(2^k)$. □

However, the difference is asymptotic only in the number of arguments, which is considered a constant because it affects only program complexity. This paper considers only data complexity, which is standard practice for data-intensive applications.

We show that the time complexity of s-topdown is no worse than the time complexity of v-topdown. For the theorems below, the time and space complexities are compared for a set of rules and a given query.

Theorem 2 (Subsumptive beats variant in time)**.** $T_{s-topdn} \leq T_{v-topdn}$.

Proof. A subquery generated during s-topdown is guaranteed to be generated during v-topdown, since they follow the same algorithm, except that s-topdown may avoid generating some subqueries. Therefore, the facts inferred during s-topdown is a subset, not necessarily proper, of the facts inferred during v-topdown. Table lookup is more costly in s-topdown as shown in Lemma 1, but it does not affect data complexity. We obtain $T_{s-topdn} \leq T_{v-topdn}$. □

Using a similar argument, we show that s-topdown uses no more space than v-topdown asymptotically.

Theorem 3 (Subsumptive beats variant in space)**.** $S_{s-topdn} \leq S_{v-topdn}$.

Proof. As described in the last proof, the subqueries generated during s-topdown is a subset of the subqueries generated during v-topdown. Therefore, the number of tables

created during s-topdown is no more than during v-topdown, and the number of answers stored in tables that exist in both are equal since they both should compute the same answers. Therefore, $S_{s-topdn} \leq S_{v-topdn}$. □

We have established that s-topdown is at worst equal to v-topdown. The following theorem shows that it can in fact be better in both time and space complexities.

Theorem 4 (Subsumptive properly beats variant)**.** *There exists a set of rules and a query for which* $T_{s-topdn} < T_{v-topdn}$ *and* $S_{s-topdn} < S_{v-topdn}$.

Proof. We prove this theorem using the running example. Recall that subsumptive binding annotation of the rules for the query with both arguments free results in rules `(1a)` and `(2a)`, whereas annotation for v-topdown results in the same two rules plus four extra rules.

We have shown that the time complexity incurred by `(2a)` is $O(\texttt{\#imm} \times (\texttt{\#rel.2/1})^2)$. However, for v-topdown, consider annotated rules `(2b)` and `(2c)`, copied below:

```
rel_bf(x,y) :- imm(u,v), rel_bb(u,x), rel_bf(v,y).
rel_bb(x,y) :- imm(u,v), rel_bb(u,x), rel_bb(v,y).
```

The local complexity for `(2c)` is $O(\texttt{\#imm})$, however, there are at least $O((\texttt{\#imm.2})^2)$ invocations to `(2c)` due to the last hypothesis of `(2c)`. Therefore, the time complexity of v-topdown is at least $O(\texttt{\#imm} \times (\texttt{\#imm.2})^2)$.

Now, notice that values for the second argument of `rel` always come from the second argument of `imm`, therefore $O(\texttt{\#imm.2})$ may asymptotically be larger than $O(\texttt{\#rel.2})$, hence $O(\texttt{\#rel.2/1})$, but not vice versa. Therefore, $O(\texttt{\#imm} \times (\texttt{\#rel.2/1})^2)$ is asymptotically smaller than $O(\texttt{\#imm} \times (\texttt{\#imm.2})^2)$. Therefore, we have proven our theorem for time complexity.

For space complexity, we have shown that the space complexity of these two rules for s-topdown is $O(\texttt{\#rel})$, which is optimal since it is only as large as the output. For v-topdown, there are $O((\texttt{\#imm.2})^2)$ table entries created for `rel_bb`, which is asymptotically larger than $O(\texttt{\#rel})$ in the worst case. Therefore, we have proven our theorem for space complexity as well. □

Using Theorems 1-3 and theorems about v-topdown in [27], we establish that s-topdown beats MST for Datalog rules in *minimal form*. First, we recall the following theorem from [27].

Theorem 5 (Variant vs. MST for minimal form)**.** *For rules in minimal form,* $T_{v-botup} = T_{v-topdn}$ *and* $S_{v-topdn} \leq S_{v-botup}$.

Combining Theorem 5 with Theorems 2 and 3, we obtain that for rules in minimal form, s-topdown beats v-bottomup.

Corollary 6 (Subsumptive beats magic sets for minimal form)**.** *For rules in minimal form,* $T_{s-topdn} \leq T_{v-botup}$ *and* $S_{s-topdn} \leq S_{v-botup}$.

Finally, we give a class of rules for which the time complexity of s-topdown is better than v-topdown and v-bottomup. The main property of this class is that there are asymptotically more subsumed subqueries than answers to such subqueries in s-topdown.

Theorem 7 (A class of rules for which subsumptive properly beats variant)**.** *Let P be a set of Datalog rules and a query. If there is a rule r in P such that (i) during s-topdown evaluation, r will be used to infer answers for a query q, and the subqueries generated by a hypothesis in r are properly subsumed by q, (ii) the pattern string of those subqueries used for the conclusion of r do not bind any variables of the first hypothesis of r, (iii) the number of facts that match those subqueries is asymptotically smaller than the number of those subqueries. Then,* $T_{s-topdn} < T_{v-topdn}$ *for P.*

Proof. We analyze a set of rules and a query that satisfy the given properties. Let Q be the set of subqueries generated as described in (i), and q be the subsuming query. r is guaranteed to match all subqueries in Q, since it matches q that subsumes them. Therefore, in v-topdown, r would be used to infer answers to each subquery in Q, but in s-topdown evaluation, subqueries in Q are discarded since a subsuming query q has already been generated.

Note that each subquery in Q contains at least one more bound argument than q, since q properly subsumes subqueries in Q. Due to (ii), that bound argument does not filter any facts of the first hypothesis of r for queries in Q during v-topdown. Therefore, the complexity of v-topdown includes the product of the number of subqueries in Q and the size of the predicate of the first hypothesis of r. The complexity of s-topdown does not include this complexity since the subqueries in Q are subsumed, and due to q, it includes (as v-topdown), the product of the *answers* to subqueries in Q and the size of the predicate of the first hypothesis. If (iii) is satisfied, then the latter is asymptotically smaller than the former. Therefore, $T_{s-topdn} < T_{v-topdn}$ for P. □

4. SUBSUMPTIVE DEMAND TRANSFORMATION FOR BOTTOM-UP EVALUATION

We have shown that s-topdown beats v-topdown and the analogous transformation MST for bottom-up evaluation. However, there has been no transformation analogous to subsumptive tabling for bottom-up evaluation. In this section, we develop *subsumptive demand transformation* (SDT) for which bottom-up evaluation of the resulting rules has no worse performance than s-topdown, and compare SDT with subsumptive tabling and MST.

4.1 Subsumptive demand transformation

Subsumptive demand transformation transforms a set of rules and a query into a new set of rules, such that the set of facts that can be inferred from the new set of rules contains only facts that would be inferred during v-topdown of the original rules. It adds new rules that define needed facts for each hypothesis in each rule, adds hypotheses to the original rules to restrict computation to infer only needed facts, and adds negated hypotheses to the rules that define needed facts to make use of subsumption. For each s-demand pattern $\langle \mathtt{p}, \mathtt{s} \rangle$, and for each rule

```
p(...)  :- h1,..., hn.
```

it generates

```
p(...)  :- d_p_s(a1,...,ak), h1,..., hn.
```

where `d_p_s` is a new predicate, called *demand predicate*, and $\mathtt{a}_1$,...,$\mathtt{a}_k$ are arguments of the conclusion that are bound

by s. The added hypotheses restrict the rules to infer only facts necessary. The demand predicates are defined as follows. For the given query, p(a_1,...,a_k)?, the following fact is generated

d_p_s(a_{b1},...,a_{bl}).

where a_{b1},...,a_{bl} are the constant arguments of the query, and s is the pattern string of the query. For each rule r generated, c :- h_0,..., h_n., and for each h_i of an IDB predicate p, the following rule is generated

```
d_p_s(a1,...,ak) :- h0,..., hi-1,
                    not d_p_s1(...), ...
```

where a_1,...,a_k are the bound arguments of h_i, s is the pattern string of h_i, and there is a negated hypothesis not d_p_s_j(...) for each pattern string s_j that *properly subsumes* s, over those arguments among a_1,...,a_k that are bound by s_i. A pattern string s_1 of length n properly subsumes a pattern string s_2 of length n iff s_1 and s_2 are different, and for each i less than n, the ith character of s_1 is either 'f' or the same as the ith character of s_2. The negated hypotheses ensure that no demand has been inferred that would correspond to a subsuming query in s-topdown.

For the rules in the running example, and the query rel(c,y)?, the s-demand patterns are ⟨rel,'bf'⟩ and ⟨rel,'bb'⟩, and subsumptive demand transformation results in the following rules:

```
rel(x,y) :- d_rel_bf(x), imm(x,y).          (1bf)
rel(x,y) :- d_rel_bf(x), imm(u,v),
            rel(u,x), rel(v,y).             (2bf)
rel(x,y) :- d_rel_bb(x,y), imm(x,y).        (1bb)
rel(x,y) :- d_rel_bb(x,y), imm(u,v),        (2bb)
            rel(u,x), rel(v,y).
d_rel_bf(c).                                (Q)
d_rel_bb(u,x) :- d_rel_bf(x), imm(u,v),     (2bf.3)
                 not d_rel_bf(u),
                 not d_rel_fb(x),
                 not d_rel_ff.
d_rel_bf(v) :- d_rel_bf(x), imm(u,v),       (2bf.4)
               rel(u,x),
               not d_rel_ff.
d_rel_bb(u,x) :- d_rel_bb(x,y), imm(u,v).   (2bb.3)
                 not d_rel_bf(u),
                 not d_rel_fb(x),
                 not d_rel_ff.
d_rel_bb(u,x) :- d_rel_bb(x,y), imm(u,v),   (2bb.4)
                 rel(u,x).
                 not d_rel_bf(u),
                 not d_rel_fb(x),
                 not d_rel_ff.
```

where each rule (Ns) is generated from rule (N) for the pattern string s, the fact (Q) is generated from the given query, and each rule (Ns.M) captures the demand due to the Mth hypothesis of rule (Ns).

Finally, each resulting rule is split into rules of two hypotheses from left to right. This transformation coupled with the bottom-up evaluation provides an implementation method with the same time complexity as and better space complexity than v-bottomup.

This transformation follows the same method as variant demand transformation, except that it uses subsumptive demand patterns and inserts the negated hypotheses for subsumption of demand facts.

Handling negation. SDT introduces negated hypotheses to rules, and negation in Datalog may be interpreted under several different semantics. To match the behavior of subsumptive tabling, we use *inflationary semantics* [12] for negation in bottom-up evaluation. It is a temporal semantics. It checks whether a fact exists at the time when the negation is encountered. In other words, when there exists a substitution of variables in the rule such that all non-negated hypotheses of the rule are facts, then the negated hypotheses under that substitution are checked whether any of them has currently been inferred as a fact. If none has been inferred as a fact, the rule is used to infer the conclusion under the substitution as a fact. We call bottom-up evaluation extended with inflationary semantics *inflationary bottom-up evaluation*.

During inflationary bottom-up evaluation, a fact may be considered false at one point, and true at a later point. We show that the implementation of the rules obtained by SDT with inflationary semantics infers the same facts of the given predicates as the rules obtained by variant demand transformation, therefore SDT preserves correctness.

Theorem 8. *Let P be a set of Datalog rules and a query, P_v be the rules obtained by variant demand transformation from P, and P_s be the rules obtained by subsumptive demand transformation from P. Then, for each predicate p in P, the bottom-up evaluation of P_v and inflationary bottom-up evaluation of P_s infer the same facts of p.*

Proof. The rules that define given predicates are the same in P_v and P_s; only rules that define demand predicates from SDT have additional negated hypotheses. Suppose a fact d of a demand predicate is inferred in bottom-up evaluation of P_v and not in the bottom-up evaluation of P_s, and that d is used to infer a fact f of a given predicate. Then, for the rule used to infer d in P_v, during bottom-up evaluation of P_s, all positive hypotheses were satisfied, but at least one of the negated hypothesis was not satisfied due to a fact d'. By definition of the rule, d' is a demand fact that corresponds to a query subsuming the query corresponding to d, therefore there exists a rule that infers f using d'. Hence, the evaluation of P_v does not infer more facts for given predicates than the evaluation of P_s.

For the other direction, since the evaluation is monotonic, the evaluation of P_s cannot infer more facts than the evaluation of P_v. □

Scheduling in bottom-up. We have shown that the order of subqueries are important for s-topdown as discussed in Section 3. Analogously, the order of demand facts inferred and considered is important in s-bottomup. In inflationary bottom-up evaluation, facts are processed in an undefined order. We modify that by considering facts for the demand predicates first, and in the order they are inferred. If no demand facts are left, then we consider facts for given predicates. We call this evaluation method *demand-first inflationary bottom-up evaluation*.

By considering demand facts first and in order, demand-first inflationary bottom-up evaluation mimics s-topdown. The only difference that remains is the retrieval order of facts of given predicates. We have not defined such an order for s-topdown either, and in practice, different systems may opt for different orders. From now on, we assume that a particular retrieval order is used for facts of given predicates,

and the same order is used during demand-first inflationary bottom-up evaluation.

For P being a set of rules and a query, we call the demand-first inflationary bottom-up evaluation of the rules resulting from SDT of P, *s-bottomup* of P.

4.2 Relationship to subsumptive tabling and MST

By using the analogy in the inferred demand facts and encountered subqueries, we first show that s-bottomup beats s-topdown in time complexity.

Theorem 9 (S-bottomup beats s-topdown in time)**.** $T_{s-botup} \leq T_{s-topdn}$.

Proof. Let P be a set of rules and a query. Let P_a be the set of rules and query after subsumptive binding annotation of P, and P_s be the set of rules obtained by SDT of P.

For a rule r in P_a of the form `p(...) :- body.`, the complexity incurred by r for $T_{s-topdn}$ is $i \times l$, where i is *the number of invocations* of r, and l is *the local complexity*, which is the product of the sizes of hypotheses.

For each such rule r, there is a rule r' in P_s of the form `p(...) :- d(...), body, negated_hypos.`, where `d(...)` is the new demand hypothesis. In s-bottomup, facts of `d` in r' are obtained from all queries of `p`, in s-topdown, whose s-demand pattern is captured by `d`. S-bottomup ceases to infer new facts for `d` in the same asymptotic time as it would take for s-topdown to reach a subquery subsuming the query corresponding to the demand of `d`, since the scheduling for s-bottomup is analogous to s-topdown. Therefore, `#d` $= i$. For $T_{s-botup}$, 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`$\times l = i \times l$. Note that the negated hypotheses do not incur additional time complexity since they require constant-time lookups in the set of facts inferred.

The only rules in P_s that do not correspond to a rule in P_a are the rules that infer facts of the demand predicates. The additional complexity incurred for $T_{s-botup}$ by each such rule is already dominated by a component of the complexity in $T_{s-topdn}$, 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 $T_{s-topdn}$.

Hence, $T_{s-botup} \leq T_{s-topdn}$. □

For space complexity, the result is the opposite.

Theorem 10 (S-topdown beats s-bottomup in space)**.** $S_{s-topdn} \leq S_{s-botup}$.

Proof. S-topdown and s-bottomup infer the same set of facts for IDB predicates in the given rules. $S_{s-topdn}$ consists of the space used by these facts in the tables. $S_{s-botup}$ also contains the size of predicates introduced when the rules are split to contain at most two hypotheses. Therefore, $S_{s-topdn} \leq S_{s-botup}$. □

We show that s-bottomup and s-topdown have the same time complexity for rules in minimal form. This is as in the case between v-topdown and v-bottomup [27]. For this purpose, we reuse the lemma from [27] below.

Lemma 11. *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.*

Theorem 12 (S-bottomup equals s-topdown for minimal form)**.** *For rules in minimal form, $T_{s-botup} = T_{s-topdn}$.*

Proof. Let P be a set of rules and a query. Let P_a be the set of rules and query after subsumptive binding annotation of P, and P_s be the set of rules obtained by SDT of P. Each rule r in P_a is of one of two forms:

(i) r has one hypothesis, and so has the form `c :- h.` In P_s, 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 $T_{s-botup}$ and by r to $T_{s-topdn}$ are both dominated by the size of the predicate of `h`, since `h` contains all variables in `d`.

(ii) r has two hypotheses, and so has the form `c :-` $\mathtt{h}_1$`,` $\mathtt{h}_2$`.` In P_s, 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 $T_{s-topdn}$, denoted $T_{s-topdn}(r)$, equals the product of the sizes of the predicates `d`, $\mathtt{h}_1$, and $\mathtt{h}_2$, and the number of facts of `d` and the number of invocations to the rule is the same as argued in Theorem 9. 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 $T_{s-topdn}(r)$. The variables of `d` must appear in `new`, because they appear in `c`, and the variables of $\mathtt{h}_1$ must appear in `new` because there are no singleton variables in the rule. Then, by the lemma above, 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 $T_{s-topdn}(r)$.

Therefore, for each complexity summand incurred by rules in P_a for $T_{s-topdn}$, there is a rule in P_s that incurs the same complexity summand for $T_{s-botup}$. Combining this with Theorem 9, which states that $T_{s-botup} \leq T_{s-topdn}$, we obtain $T_{s-botup} = T_{s-topdn}$. □

To compare s-bottomup and v-bottomup, note that Theorem 8 implies that s-bottomup beats v-bottomup in space complexity, because both infer the same facts for the given predicates per Theorem 8 and s-bottomup may infer fewer facts for demand predicates.

For time complexity, s-bottomup beats v-bottomup due to Theorem 12 and Corollary 6, and is asymptotically faster for the class of rules and queries described in Theorem 7 due to the aforementioned results. Therefore, we obtain the following corollary.

Corollary 13. $T_{s-botup} \leq T_{v-botup}$ *and* $S_{s-botup} \leq S_{v-botup}$.

For the rules described in Theorem 7, $T_{s-botup} < T_{v-botup}$ and $S_{s-botup} < S_{v-botup}$.

5. SUBSUMPTION OPTIMIZATION

We have shown that s-topdown beats v-topdown, and s-bottomup beats v-bottomup. However, there are cases when the subsumptive methods are not effective because subqueries that subsume others appear later during evaluation, and the complexity may have been reduced if they had appeared earlier. In this section, we show a transformation method to ensure that subqueries that subsume others

for s-topdown and s-bottomup are processed first. We first show the effectiveness of the method for s-topdown, and then show that it works for s-bottomup as well. We call this transformation *subsumption optimization*.

The method first identifies demand patterns that should be subsumed, and then transforms the rules so that any subquery with such a demand pattern is subsumed during s-topdown.

(i) Identification of demand patterns to subsume. For a set s of demand patterns of a predicate determined for the given rules and query, the method generates all sets of subsuming demand patterns, and generates annotated rules for each such set. Then, for each set of the resulting rules, we compare the time complexity of the resulting annotated rules with the original annotated rules. If it can be proven that the asymptotic time complexity of the resulting rules due to a set s_2 of subsuming demand patterns is lower than the original annotated rules, then we say that the demand patterns in s should be subsumed by the demand patterns in s_2.

For rules (1) and (2) in the running example, and the query `rel(c,y)?`, the demand patterns are ⟨`rel`,'`bf`'⟩ and ⟨`rel`,'`bb`'⟩. The rules obtained by subsumptive binding annotation are:

```
rel_bf(x,y) :- imm(x,y).
rel_bf(x,y) :- imm(u,v), rel_bb(u,x), rel_bf(v,y).
rel_bb(x,y) :- imm(x,y).
rel_bb(x,y) :- imm(u,v), rel_bb(u,x), rel_bb(v,y).
```

It would be asymptotically better in time complexity if the calls to `rel_bb` were subsumed by previously encountered `rel_bf` subqueries. The intuition is that if the outdegree of the `rel` relation is not constant, then asymptotically many more queries to `rel_bb` may be made than the ones that would be relevant due to the last rule. Therefore, the call with pattern ⟨`rel`,'`bb`'⟩ should be subsumed by the call with pattern ⟨`rel`,'`bf`'⟩.

(ii) Transformation. For each s-demand pattern ⟨p, s⟩ for calls that should be subsumed by calls with pattern ⟨p, s2⟩, for each hypothesis $\texttt{p}(\texttt{a}_1,\ldots,\texttt{a}_n)$ whose pattern string is `s` in a rule, we generate the following rule:

$\texttt{q}(\texttt{a}_{b1},\ldots,\texttt{a}_{bk})$ `:-` $\texttt{p}(\texttt{a}_1,\ldots,\texttt{a}_n)$.

where `q` is a fresh predicate name, and $\texttt{a}_{b1},\ldots,\texttt{a}_{bk}$ are those arguments among $\texttt{a}_1,\ldots,\texttt{a}_n$ that are bound by `s2`. Then, before this hypothesis in r, we insert $\texttt{q}(\texttt{a}_{b1},\ldots,\texttt{a}_{bk})$ as a hypothesis. After this transformation, a subquery that subsumes the subquery by the original ith hypothesis $\texttt{p}(\texttt{a}_1,\ldots,\texttt{a}_n)$ will be guaranteed to be made first during s-topdown, so all subqueries with the demand pattern ⟨p, s⟩ will be avoided.

For the second rule in the running example, the second hypothesis is the only hypothesis whose pattern string '`bb`' should be avoided. Thus, this rule is transformed into the following two rules.

```
new(u) :-  rel(u,x).
rel_bf(x,y) :- imm(u,v), new(u), rel(u,x), rel(v,y).
```

We prove that subsumption optimization preserves the semantics of the original set of rules.

Theorem 14. *Let P be a set of Datalog rules and a query. For each given predicate of P, s-topdown of P and s-topdown of the rules resulting from subsumption optimization infer the same facts.*

Proof. The rules after subsumption optimization are more restricted due to added hypotheses, so s-topdown of those rules cannot infer more facts than s-topdown of P. However, they cannot infer fewer either, since the added hypotheses are defined by rules whose hypothesis subsume existing hypotheses, therefore they cannot be false while all other hypotheses are true. Hence, s-topdown of P and s-topdown of the rules after subsumption optimization infer the same facts for given predicates. □

We show that this transformation achieves the same effect for s-bottomup, by showing that the introduced hypotheses and rules ensure that demand facts cannot be inferred for demand patterns that the optimization determines to subsume.

Theorem 15. *Let P be a set of Datalog rules and a query. The s-bottomup of P after subsumption optimization does not infer any fact for the demand predicates that correspond to the demand patterns that the optimization determines to subsume.*

Proof. For any hypothesis h with a demand pattern that the optimization determines to subsume, the transformation introduces a new hypothesis to its left. For that hypothesis to be true, the demand fact that corresponds to a subsuming demand pattern must be inferred by construction of the transformation. Thus, when all of the hypotheses to the left of h is true, a demand fact corresponding to a demand pattern subsuming the demand pattern for h must have been inferred. Therefore, no fact can be inferred for demand predicates of demand patterns that the optimization determines to subsume. □

Application to demand-driven pointer analysis. We show that by applying subsumption optimization to the specification of Andersen's pointer analysis for C [2], we automatically derive Heintze and Tardieu's algorithm for demand-driven pointer analysis [11], and our complexity analysis can be used to obtain precise time and space complexities.

Given a C program, statements in a program relevant to pointer analysis can be reduced to four kinds, which can be represented directly as Datalog facts:

- `p = &q` is represented by `bare_addr(p,q)`.
- `p = q` is represented by `bare_bare(p,q)`.
- `p = *q` is represented by `bare_star(p,q)`.
- `*p = q` is represented by `star_bare(p,q)`.

Andersen's pointer analysis can be specified directly using four Datalog rules, where `pt(p,q)` denotes `p` points to `q`:

```
pt(p,q) :- bare_addr(p,q).                    (A1)
pt(p,q) :- bare_bare(p,r), pt(r,q).           (A2)
pt(p,q) :- bare_star(p,s), pt(s,r), pt(r,q). (A3)
pt(p,q) :- star_bare(r,s), pt(r,p), pt(s,q). (A4)
```

Given a query `pt(c,q)?`, the s-demand patterns are ⟨`pt`,'`bf`'⟩ and ⟨`pt`,'`bb`'⟩. Subsumptive binding annotation yields:

```
pt_bf(p,q) :- bare_addr_bf(p,q).
pt_bf(p,q) :- bare_bare_bf(p,r), pt_bf(r,q).
pt_bf(p,q) :- bare_star_bf(p,s), pt_bf(s,r),
              pt_bf(r,q).
pt_bf(p,q) :- star_bare_ff(r,s), pt_bb(r,p),
              pt_bf(s,q).

pt_bb(p,q) :- bare_addr_bb(p,q).
pt_bb(p,q) :- bare_bare_bf(p,r), pt_bb(r,q).
pt_bb(p,q) :- bare_star_bf(p,s), pt_bf(s,r),
              pt_bb(r,q).
pt_bb(p,q) :- star_bare_ff(r,s), pt_bb(r,p),
              pt_bb(s,q).
```

Our complexity analysis can be used to show that asymptotically more queries to `pt_bb` may be made than the ones that would be relevant due to the last two rules generated. The number of calls to `pt_bb` may be as many as v^2 where v is the number of variables in the C program. However, the `pt` relation may not be as dense as $O(v^2)$. Due to this fact, the complexity of the last rule defining `pt_bb` is $O(\texttt{\#star\_bare\_ff} \times v^2)$. This is greater than the complexity of the last rule defining `pt_bf`, which is $O(\texttt{\#star\_bare\_ff} \times \texttt{\#pt.2/1} \times v)$. Therefore, $\langle \texttt{pt}, \texttt{'bb'} \rangle$ should be subsumed by $\langle \texttt{pt}, \texttt{'bf'} \rangle$ so that the last four rules are not generated. Subsumption optimization results in the rules `(A1)`, `(A2)`, `(A3)`, and the following two rules due to `(A4)`:

```
new(r) :- pt(r,p).
pt(p,q) :- star_bare(r,s), new(r), pt(r,p), pt(s,q).
```

S-bottomup of the resulting rules corresponds precisely to Heintze and Tardieu's algorithm, and performing precise complexity analysis gives the explanation for when and why the algorithm is efficient.

6. EXPERIMENTS

We support our complexity analyses and comparisons by experiments. For s-topdown and v-topdown, we use XSB [6]. For v-bottomup and s-bottomup, we use the implementation method of [16] modified with the described extensions when necessary to generate Python code from the rules.

We instantiate the complexity parameters in predicted complexities with their values computed from the data. We use *space units* to mean the number of unique table inserts for v- and s-topdown, and the number of facts inferred plus the number of elements in auxiliary maps [16] for v- and s-bottomup. We use *returns* to mean the number of facts returned from rule invocations for tabled top-down evaluation, and *firings* to mean the number of firings for demand-driven bottom-up evaluation.

In our benchmarks, predicates have two arguments. For experiments, we first fix `#p` and `#p.1/2` for each input predicate `p`, and 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.

We first show experimental results for the running example and the demand-driven pointer analysis problem.

For a query with both arguments free for the running example, we confirm the asymptotic time and space complexities analyzed. The left plot in Figure 1 for the running example shows that s-topdown and s-bottomup are asymptotically faster than v-topdown and v-bottomup, respectively, as predicated by Theorems 2 and 4, and by Corollary 13, respectively. It also shows that s-bottomup is asymptotically faster than s-topdown, due to splitting of rules into two hypotheses each, as predicated by Theorem 9.

The right plot in Figure 1 for the running example shows that the space usage of v- and s-topdown is much smaller than v-bottomup and s-bottomup, respectively. This is because s-bottomup infers facts for demand predicates and the predicates introduced for splitting rules, and maintains auxiliary maps. Demand predicates are avoided in top-down evaluation by backtracking, which uses stack space that this paper does not consider. Predicates introduced for splitting rules allow s-bottomup to be faster than s-topdown as shown in Theorem 9. Auxiliary maps are avoided in s-topdown by separate indexing mechanisms (such as tries in XSB), whose space is not counted separately in this paper. The plot also shows that the space usage of s-topdown and s-bottomup is smaller than v-topdown and v-bottomup, respectively, due to fewer facts being inferred, as shown in Theorems 3 and 4, and in Corollary 13, respectively

For demand-driven pointer analysis, and a query with the first argument bound, we apply subsumption optimization to the rules and confirm the asymptotic improvement in time complexity against s-bottomup, and show the difference in space usage. Figure 2 for demand-driven pointer analysis confirms that subsumption optimization improves time complexity and reduces space usage.

We also performed experiments on a well-known benchmark in semantic web [15] that has 961 rules and 654 facts; and we also performed the pointer analysis on real data from a C program with 2430 facts. Both queries run out of memory in XSB when v-topdown is used on a machine with 4 GB of RAM, but when s-topdown is used, the ontology query runs in 12 seconds, and the pointer analysis runs in under 0.1 seconds.

Finally, to compare our results with existing SQL database implementations, we converted the rules of the running example into SQL queries and performed experiments on MSSQL, one of the few SQL databases that support recursion. The running example timed out (more than 10 minutes) on all of our data points except for the smallest two. The variant and subsumptive demand transformations cannot be used on SQL databases because no mainstream SQL database implementation properly supports mutual recursion, and demand transformations result in mutually recursive rules.

7. RELATED WORK AND CONCLUSION

Datalog has been extensively studied [5, 1]. Variant tabling was introduced in the 1980s [25], and has been widely studied. It has been implemented in top-down evaluation engines such as XSB [6] and YAP [8]. Subsumptive tabling was introduced ten years later [21], but has not been studied widely, and has only been implemented in XSB.

The time and space complexity of variant tabling for Datalog was given an imprecise upper bound in [29], and a method for precise calculation of the time and space complexities of variant tabling for Datalog is given in [27]. There is no prior work on complexity analysis of subsumptive tabling to the best of our knowledge. Our work builds on [27] to present the first method for precise calculation of worst-case time and space complexities of subsumptive tabling, and establishes that it beats variant tabling.

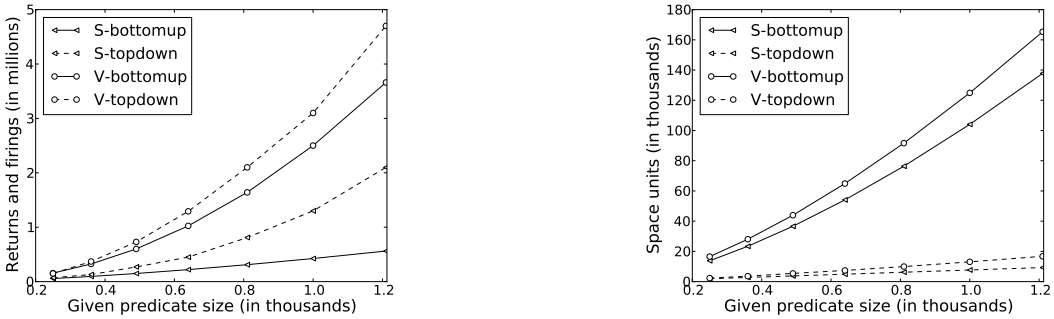

Figure 1: Firings/returns and space units for v-topdown, v-bottomup, s-topdown, and s-bottomup for the running example.

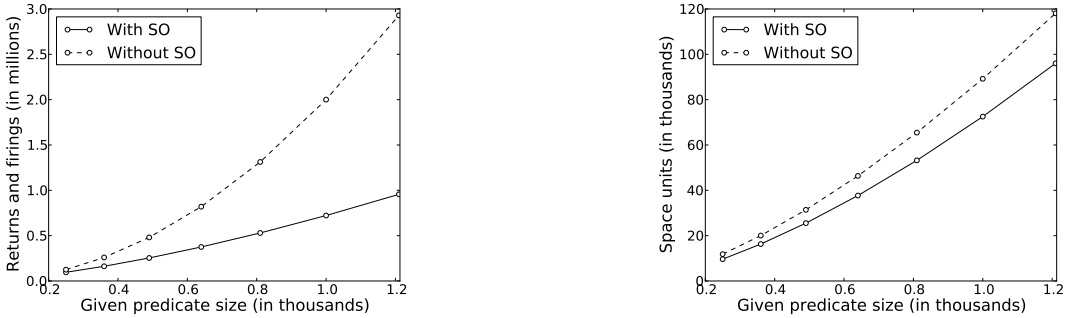

Figure 2: Firings/returns and space units for s-bottomup with and without subsumption optimization (SO) for the pointer analysis benchmark.

For bottom-up evaluation, transformations for demand-driven evaluation have been studied, including the well-known magic-set transformation (MST) [3] and the recent demand transformation [27] that our work extends. It has been shown that both of these transformations are equivalent to variant tabling both operationally [4] and in terms of complexities [27]. We show that subsumptive tabling beats MST, and then give the first transformation, subsumptive demand transformation, so that bottom-up evaluation of resulting rules is similar to subsumptive tabling. We show that the complexity of the resulting rules from this new transformation beats subsumptive tabling and MST in time complexity.

The relationship between top-down and bottom-up evaluation has been studied in a variety of contexts with different flavors of rules [20, 28, 19, 4]. Our work is the first to establish precise relationships between variant and subsumptive tabling, and MST and the new subsumptive demand transformation.

By characterizing improvement using complexity analysis, we also introduce a new transformation, subsumption optimization, to reuse as many facts as possible. We have shown that our methods can be used to systematically derive a well-known demand-driven pointer analysis algorithm [11].

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

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] L. O. Andersen. Program analysis and specialization for the C programming language. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1994.

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

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

[5] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990.

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

[7] S. S. Cosmadakis, H. Gaifman, P. C. Kanellakis, and M. Y. Vardi. Decidable optimization problems for database logic programs (preliminary report). In *Proc. of the 20th Annual ACM Symp. on Theory of Computing*, pages 477–490, 1988.

[8] A. F. da Silva and V. S. Costa. The design of the YAP compiler: An optimizing compiler for logic programming languages. *J. of Universal Computer Science*, 12(7):764–787, 2006.

[9] J. DeTreville. Binder, a logic-based security language.

In *Proc. of the 2002 IEEE Symp. on Security and Privacy (S&P)*, pages 105–113, 2002.

[10] E. Hajiyev, M. Verbaere, and O. 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.

[11] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 24–34, 2001.

[12] P. G. Kolaitis and C. H. Papadimitriou. Why not negation by fixpoint? *J. Comput. Syst. Sci.*, 43(1):125–144, 1991.

[13] A. Y. Levy and Y. Sagiv. Semantic query optimization in Datalog programs. In *Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS)*, pages 163–173, 1995.

[14] N. Li and J. 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.

[15] S. Liang, P. Fodor, H. Wan, and M. 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.

[16] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Trans. Programming Languages and Systems*, 31(6):21:1–21:38, August 2009.

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

[18] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proc. of the 5th Intl. Conf. and Symp. on Logic Programming*, pages 140–159, 1988.

[19] R. 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.

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

[21] P. Rao, C. R. Ramakrishnan, and I. V. Ramakrishnan. A thread in time saves tabling time. In *Proc. of the 1996 Joint Intl. Conf. and Symp. on Logic Programming*, pages 112–126, 1996.

[22] Y. Sagiv. Optimizing Datalog programs. In *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, 1988.

[23] D. Sereni, P. Avgustinov, and O. de Moor. Adding magic to an optimising Datalog compiler. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD)*, pages 553–566, 2008.

[24] W. Shen, A. Doan, J. F. Naughton, and R. 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.

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

[26] K. T. Tekle, M. Gorbovitski, and Y. A. Liu. Graph queries through Datalog optimizations. In *Proc. of the 12th Intl. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP)*, pages 25–34, 2010.

[27] K. T. Tekle and Y. A. Liu. Precise complexity analysis for efficient Datalog queries. In *Proc. of the 12th Intl. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP)*, pages 35–44, 2010.

[28] J. 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.

[29] D. S. Warren. Programming in tabled Prolog. Available at http://www.cs.sunysb.edu/~warren/xsbbook/, 1999.

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