Space Optimization in the Bottom-Up Evaluation of Logic Programs

S. Sudarshan Divesh Srivastava Raghu Ramakrishnan*
Jeffrey F. Naughton†
{sudarsha,divesh,raghu,naughton}@cs.wisc.edu
Computer Sciences Department,
University of Wisconsin-Madison, WI 53706, U.S.A.

Abstract

In the bottom-up evaluation of a logic program, all generated facts are usually assumed to be stored until the end of the evaluation. Considerable gains can be achieved by instead discarding facts that are no longer required: the space needed to evaluate the program is reduced, I/O costs may be reduced, and the costs of maintaining and accessing indices, eliminating duplicates etc. are reduced. Thus, discarding facts early could achieve time as well as space improvements.

Given an evaluation method that is sound, complete and does not repeat derivation steps, we consider how facts can be discarded during the evaluation without compromising these properties. Our first contribution is to show that such a space optimization technique has three distinct components. Informally, we must make all derivations that we can with each fact, detect all duplicate derivations of facts and try to order the computation so as to minimize the "lifespan" of each fact.

This separation enables us to use different methods for each of the components for different parts of the program. We present several methods for ensuring each of these components. We also briefly describe how to obtain a complete space optimization technique by making a choice of techniques for each component and combining them. Our results apply to a significantly larger class of programs than those considered in [NR90].

*The work of the first three authors was supported in part by a David and Lucile Packard Foundation Fellowship in Science and Engineering, an IBM Faculty Development Award and NSF grant IRI-8804319.

†The work of this author supported by NSF grant IRI-8909795.

1 Introduction

Bottom-up evaluation of a logic program proceeds by repeatedly applying rules to generate facts until no new facts can be produced. Bottom-up evaluation has been shown to have several advantages over top-down evaluation in the area of deductive databases (see, for example, [Ull89]). However, a disadvantage of bottom-up evaluation is that all generated facts are usually assumed to be stored until the end of the evaluation. Since the number of facts generated can be extremely large in the case of many programs, reducing the space requirements of a program by discarding facts during the evaluation may be very important. The following example (from [NR90]) illustrates this point.

Example 1.1 The program computes the length of the longest common subsequence (LCS) of two strings a and b, and is representative of more general sequence analysis programs. The algorithm is from Hirschberg [Hir75]; we use the representation that if letter j of string a (resp. b) is α, then the database contains the fact $a(j, \alpha)$ (resp. $b(j, \alpha)$).

$R1 : lcs(m, N, 0).$
$R2 : lcs(M, n, 0).$
$R3 : lcs(M, N, X) \leftarrow M < m, N < n, a(M, C), b(N, C), lcs(M+1, N+1, X-1).$
$R4 : lcs(M, N, X) \leftarrow M < m, N < n, a(M, C), b(N, D), C <> D, lcs(M+1, N, X1), lcs(M, N+1, X2), X = \max(X1, X2)$
Query: $?\text{-}lcs(0, 0, X).$

If the strings are of both of length n, then evaluating the program using the top-down Prolog evaluation strategy gives a running time that is $\Omega(\binom{2n}{n})$. Using the Magic Sets rewriting strategy followed by bottom-up evaluation produces a running time that is $O(n^2)$, but is also space $\Omega(n^2)$. Suppose n is 10^6, a value that we are likely to see in applications such as DNA sequencing. Then the number of facts generated is around 10^{12}, which is clearly impractical to store.

Sliding Window Tabulation, as described in [NR90], evaluates this program in $O(n)$ space and $O(n^2)$ time, by discarding facts in the course of the computation. Storing 10^6

facts is certainly feasible, as opposed to 10^{12}. Thus this improvement in the space complexity is essential if the program is to be run on large data sets. □

In addition to improving the space requirements, discarding facts that are no longer needed can have other advantages. I/O costs may be reduced, even eliminated, if the program can be evaluated in main memory; the costs of maintaining and accessing indices, eliminating duplicates etc. are also reduced. Thus, discarding facts early could achieve time as well as space improvements. We refer to techniques that discard facts during the course of the evaluation of a logic program as *space optimization techniques*[1].

While Sliding Window Tabulation is effective on the LCS example, the applicability of Sliding Window Tabulation as presented in [NR90] is fairly limited. For instance, suppose we extend the LCS program so that instead of being base predicates, a and b are defined by additional rules in the program—this will be the case if the program preprocesses "rough" base data before searching for common subsequences. Sliding Window Tabulation cannot be used on this extension of the LCS program. Similarly, if the above program is embedded in a larger program that uses the length of the longest common subsequence to perform further analysis, such as find the region of this DNA sequence that best matches the given test sequence, Sliding Window Tabulation is again inapplicable.

One of the main contributions of this paper is to show that space optimization techniques have three components:

1. Ensuring that each fact is used in every possible derivation even though facts may be discarded before the end of the evaluation.
2. Ensuring that multiple derivations of a fact are detected, in order to avoid repeated inferences.
3. Synchronizing the computation to ensure that derivations of facts are "close" to all their uses, and discarding facts soon after their uses.

We describe these components in more detail in Sections 2. This decomposition provides a framework in which to reason about space optimization methods. It also gives us the flexibility of choosing different techniques for each component, and synthesizing new space optimization techniques.

In this paper, we also describe several new techniques for each of these three components, and briefly discuss how to combine these techniques to generate space optimization methods. (Automatically combining these techniques to get a space optimization technique for the full program is discussed in detail in the full version of this paper [SSRN90].)

[1] In this paper, we do not consider other space saving techniques, such as allowing facts to share parts of their structure with other facts.

Sliding Window Tabulation then turns out to be just one particular way of combining techniques for each of the components. Thus our techniques significantly extend the class of programs optimized in [NR90]. In particular, we can deal with some programs (rewritten using Magic Sets) in which a predicate p and the corresponding magic predicate $magic\_p$ are mutually recursive whereas the techniques of [NR90] do not handle such programs.

1.1 Definitions

In this paper, we consider Horn clause logic programs[2], and assume the usual definitions including those of terms, atoms and rules (clauses). We assume familiarity with semi-naive evaluation, and in some of the sections of the paper we also assume some familiarity with the Magic Sets transformation. We refer the reader to [Ull89] for an introduction.

A *program* is treated as a set of rules and EDB facts. While analyzing the program, we do not need to know the specific EDB facts, but we often make use of information such as functional dependencies on EDB relations. A *program fact* is used to mean any fact that is used or derived by a program. In this paper, we assume that all program facts are ground. We say that a predicate is *derived* with respect to another if the two predicates are mutually recursive[3]. We say that a predicate is *derived wrt* a rule if it is derived wrt the head predicate of the rule. A derived predicate occurrence in a rule is an occurrence in the rule of a predicate that is derived wrt the rule.

In the rest of this paper, we use the terms 'derivation of a fact,' '(point in) the evaluation of a program,' etc. without formal definitions. They are what one would intuitively expect in a bottom-up evaluation, and formal definitions are included in the full version of this paper [SSRN90]. We assume that the program is evaluated using semi-naive evaluation[4].

2 Soundness, Completeness and Non-Redundancy

In general, discarding a fact could result in the non-derivation of other facts that should have been derived and thereby, in the presence of negation, derivation of facts that should not have been derived. This could compromise completeness, and in the presence of negation, also soundness. The following condition ensures that facts are used in all possible derivations and is used to ensure soundness and completeness:

[2] This can be extended to include programs with stratified negation and aggregation.

[3] We refine and extend this definition in the full version of this paper.

[4] In the full version of the paper, we consider a synchronization technique called Nested-SCC evaluation that could result in some repeated derivations.

Condition U: A fact $p(\overline{a})$ satisfies Condition U with respect to an evaluation, at a point $e1$ in the evaluation, iff

1. Every derivation using it has been made at or before $e1$, or
2. If $p(\overline{a})$ is discarded at $e1$, it will be recomputed at some later point $e2$ in the evaluation, and any derivation that could have been made using $p(\overline{a})$ after $e1$ (had $p(\overline{a})$ not been discarded) will be made after the fact has been recomputed at $e2$. Also, if the program has negation, any derivation that would have been prevented by the presence of $p(\overline{a})$ is not made between $e1$ and $e2$. □

In this paper, we assume that an evaluation is a semi-naive evaluation [BR87, RSS90]. Such an evaluation has the desirable property of not repeating inferences if no facts are discarded. However, if a fact is discarded and subsequently rederived, we may not detect this duplicate derivation and thus may repeat some derivations that use this fact. This could compromise the semi-naive property. Further, not detecting duplicate derivations of a fact could compromise termination if cyclic derivations are possible. The following condition ensures that multiple derivations of a fact are detected:

Condition D: A fact $p(\overline{a})$ satisfies Condition D with respect to an evaluation, at a point $e1$ in the evaluation, iff

1. It is not derived again at or after the point $e1$, or
2. If $p(\overline{a})$ is discarded at $e1$, then for any later point $e2$ in the evaluation where $p(\overline{a})$ is derived, no inference using $p(\overline{a})$ made at or before $e1$ is repeated after $e2$. □

We say that an evaluation is *derivation-complete* if all derivations that can be made using the rules and facts are in fact made by the evaluation.

Theorem 2.1 *Consider an evaluation E of a program such that E is sound, derivation-complete, has the semi-naive property and no facts are discarded during the evaluation. Suppose we modify E by discarding facts one at a time during the evaluation. Then the modified evaluation is sound, derivation-complete and has the semi-naive property iff Conditions U and D are satisfied by each fact whenever it is discarded.* □

In the rest of this paper, we assume that facts are discarded one at a time, and Conditions U and D reflect this assumption.

Theorem 2.2 *Given a logic program P, an EDB $D1$, and an arbitrary point $e1$ in an evaluation of program P on $D1$, it is undecidable whether a given fact satisfies Conditions U and/or D at $e1$.*[5] □

[5] For Datalog programs, this is decidable, but we may have to evaluate the program fully in order to check the conditions, which defeats the aim of discarding facts during the evaluation of the program.

Consequently, it is undecidable whether discarding a fact during an evaluation will compromise the soundness, completeness or semi-naive property of the evaluation. Hence, we must look for sufficient conditions for ensuring D and U for program facts. Even the stronger conditions that only test the first parts of Conditions U and D are undecidable. Our sufficient conditions are often based on the first parts of Conditions D and U.

2.1 An Overview of Our Approach

We now present a brief overview of our techniques. In subsequent sections we look in detail at some of the techniques that we outline here. Consider facts of the form $p(\overline{a})$ in a program P. Consider an evaluation of P and let ψ be a schedule for discarding p-facts in this evaluation. We justify ψ by establishing that Conditions D and U hold for each p fact before it is discarded. At compile time we analyze the program, and decide on the applicability of each technique. We then add extra tests and auxiliary computations (that we describe along with each technique) to the compiled version of the program. In general these tests are performed at run time to decide when a fact satisfies Conditions D and U. These operations are quite efficient—see Section 7 for more details. Facts are discarded at run-time as soon as the tests determine that they satisfy both Conditions D and U.

Ensuring Condition D : Condition D can be checked on a per-rule basis, and different techniques can be used for different rules in a given program. Applicable techniques include the following:
(1) Providing a bound on the total number of derivations of a fact: If a program is duplicate free ([MR90]), we know that once a fact is derived it will not be derived again. We look at this technique for ensuring Condition D in Section 3.
(2) Using monotonicity constraints: Monotonicity constraints ensure some monotone ordering on the derivation of facts. We look at this idea in Section 4.1.

Ensuring Condition U : Condition U can be checked on a per-body-literal basis, and different techniques can be used for different literals in a given program. Applicable techniques include the following:
(1) Providing a bound on the total number of uses of a fact: Suppose a rule in linear, i.e. there is only one predicate in the body of the rule which is mutually recursive with the head of the rule. Once a fact for the derived predicate is used (with all the facts for the base predicates), we know that no new derivations can be made using that fact in that rule. We look at this and more general ways of ensuring Condition U in Section 3.
(2) Using monotonicity constraints: In Section 4.2 we consider using monotonicity constraints to satisfy Condition U.

If none of these approaches for ensuring D or U succeeds, we always have the option of not discarding any p-facts. We can still optimize the rest of the program, unlike the technique described in [NR90].

Synchronization : If (all) derivations of facts are "close" to all their uses, facts can be discarded soon after being derived. We call techniques that can be used to order a computation so as to maximize this property *synchronization* techniques. These include:

(1) Interleaved-SCC synchronization: This technique is described in detail in Section 5.

(2) Delaying first use of facts: The goal is to balance the derivation of new facts against the identification of facts that can be discarded so that the number of facts that are stored at any one time is minimized. The set of derived facts is partitioned into a set of "active" facts used in derivations and a set of "hidden" facts whose use is delayed.

Although this idea is present as a part of Sliding Window Tabulation [NR90], we isolate the synchronization achieved by hiding facts from other components of space optimization techniques.

(3) Nested-SCC synchronization: This technique can be understood as identifying "subgoals" that are to be evaluated by a "subprogram" on each call. The idea is to generate facts for the subprogram as and when they are needed by the main program.

We omit the last two techniques from the short version of this paper due to lack of space.

Combining Techniques : The various techniques for synchronization and for ensuring Conditions D and U are applicable to parts of a program (such as rules, predicate occurrences, etc). These need to be combined to get a space optimization technique for the full program. We briefly look at this in Section 6.

3 Bounds on Derivations and Uses of Facts

Condition DF1: (1) No fact for p is derived by more than one rule, (2) there is at most one derivation for each p fact by any rule, and (3) no derivation for any p fact is repeated. □

The techniques of [MR90] can be used to test the first two parts of this condition. The third part of the condition is satisfied in a semi-naive evaluation if duplicate derivations of all other facts are detected (using any of the techniques described for ensuring Condition D).

Proposition 3.1 If a predicate p in a semi-naive evaluation satisfies Condition DF1, Condition D is satisfied by each p fact after it is derived. □

The essential idea is that no fact for p is derived more than once in this evaluation (i.e., p is duplicate free). We can weaken Condition DF1 in several ways. Suppose requirement (1) does not hold, we can still ensure Condition D using a run-time check to determine that a fact has been derived once by every rule that could possibly derive it. We can also use functional dependencies between the head of a rule and derived predicates in the rule body to bound the number of rule applications that derive a fact. We discuss these extensions in the full version of this paper.

Condition Bounds_U: Consider an evaluation of a program P, and a rule R : $p2() \leftarrow p(\bar{t}), b(), p1()$., where $b()$ denotes the join of all the base predicate occurrences (other than $p(\bar{t})$, if it is base) in the body of the rule, and $p1()$ denotes the join of all derived predicate occurrences (other than $p(\bar{t})$) in the body of R. Suppose that no derivations for any $p2$ facts are repeated. Then the predicate occurrence $p(\bar{t})$ in R satisfies Condition Bounds_U if $p(\bar{t})$ functionally determines $p1()$ in R. □

Note that in the case when $p(\bar{t})$ is the only predicate occurrence that is derived with respect to the rule (i.e. R is a "linear" rule), Bounds_U is trivially satisfied.

Proposition 3.2 Consider an evaluation of a program, and let a body predicate occurrence $p(\bar{t})$ in R satisfy Condition Bounds_U. A fact $p(\bar{a})$ can no longer be used in the occurrence $p(\bar{t})$ if: (1) $p(\bar{a})$ does not match any facts for $b()$, or (2) $p(\bar{a})$ matches base facts, and it has been used in the occurrence $p(\bar{t})$ in a successful rule application. □

Suppose some derivations using R and $p(\bar{a})$ are repeated. If we discard the fact $p(\bar{a})$ after one successful derivation, we would prevent repetitions of that derivation and hence not satisfy Condition U. Condition BU1 can be generalized by requiring that $p(\bar{t})$ along with $b()$ functionally determine $p1()$ in R. We examine this and other generalizations in the full version of this paper.

Example 3.1 Consider the program below.

$$anc(X,Y) \leftarrow father(X,Y).$$
$$anc(X,Y) \leftarrow father(X,Z), anc(Z,Y).$$

Suppose we know that the $father$ relation is a tree. That is, we are given the following functional dependency: $father$: $1 \rightarrow 2$, i.e. each person has only one father, and we also know that the $father$ relation is acyclic. Suppose also that the user gives a query $? - anc(X,Y)$. We assume that if a fact is an answer to the query, it is printed out straight away to the users terminal.

We can deduce the following about the program:

- The program is duplicate free: The techniques of [MR90] may be used to deduce this. Informally, this is because the functional dependency shows that each fact can be deduced at most once by each rule, and the

acyclicity of *father* along with the functional dependency shows that each fact can be deduced by at most one rule.

- Since the rule is linear, each derived fact for *anc* is used in at most one rule application. We can then discard *anc* facts once they have been used in one rule application.

We now look at the benefits due to discarding facts in this program. Let n be the number of facts in the relation *father*. The functional dependency on *anc* shows that for each node x, there is exactly one ancestor at each distance i. This means that at each stage, at most n facts are computed. Thus at most $2 * n$ *anc* facts are stored at any point of time. If facts were not discarded, up to $O(n^2)$ facts may need to be stored, depending on the structure of the *father* relation. Note that monotonicity based techniques (such as Sliding Window Tabulation and extensions discussed in Section 4) are not applicable to this program, since there is no monotonicity inherent in the rules.

Even in the absence of the functional dependency and acyclicity information of the above form, facts for the *anc* program can be deduced to satisfy Conditions D and U. Suppose for some a, no fact $anc(\_, a)$ (for any value "_") is derived in a particular iteration in a semi-naive evaluation of this program, then facts of the form $anc(\_, a)$ will neither be derived again nor used again in the evaluation. Therefore all such facts satisfy Conditions D and U, and can be discarded (if they are not answers to the query). We expect the savings to be significant in practise, although we cannot make any stronger claims about it. This idea can be extended to more general classes of programs, to derive other techniques for ensuring Condition D, but we do not pursue it here. □

4 Monotonicity

In this section we look at how to use monotonicity to ensure Conditions D and U. Our results on the use of monotonicity extend the results of [NR90].

We make extensive use of the ϕ function defined in [NR90]. The function ϕ can take any predicate p as an argument and returns an arithmetic expression involving only the argument positions of p. Further, ϕ can also take any fact $p(\bar{a})$ as an argument, and returns an integer. Candidate functions for ϕ can be generated using the techniques discussed in [NR90]. We do not discuss this here, but assume that possible ϕ functions are made available. The function ϕ has a natural extension that also can take as argument an atom $p(\bar{t})$, and returns an arithmetic expression involving only the variables in $\bar{t}$. For instance, a ϕ that maps $fac(I, N)$ to I would map $fac(I + 1, N1)$ to $I + 1$. Such a ϕ also maps $fac(4, \_)$ to 4.

4.1 Monotonicity and Condition D

Definition 4.1 Locally Saturated : Suppose that no derivation for any p fact is repeated in an evaluation. Then a set of facts for derived body predicate occurrences of a rule R defining p is said to be *locally saturated with respect to* R if every derivation that can be made using (1) R, (2) all facts for the base predicate occurrences of R, and (3) only the given set of facts for derived body predicate occurrences of R, has already been made. A set of facts is said to be *locally saturated with respect to a set of rules* if it is locally saturated with respect to each of the rules. □

Since all derivations that could be made using just the set of locally saturated facts have been made, any new derivation requires at least one fact (for a derived predicate occurrence) that is not in the set of locally saturated facts.

In the case of the Semi-Naive evaluation of an SCC (where the set of predicates derived with respect to p is just the set of predicates defined in the SCC of p), at any point in the $n+1$th iteration, the set of facts derived before the nth iteration is a set of locally saturated facts for p. If a different evaluation or synchronization technique is used, the sets of locally saturated facts may change, but the following results would not be affected. Thus we achieve a certain degree of independence from evaluation techniques in the following results.

Definition 4.2 Monotonicity : A rule is said to be *monotonically increasing with respect to a predicate occurrence* p' in its body if, for every instance of the rule (with $p(\bar{b})$ used in p'), $\phi(head) \geq \phi(p(\bar{b}))$. A rule is said to be *monotonically increasing* if it is monotonically increasing with respect to each body occurrence of a predicate that is derived with respect to the rule. □

The following is a sufficient algorithmic test for monotonicity. Consider a rule R:

$$R : p(\bar{t}) \leftarrow p_1(\bar{t_1}), \ldots, p_n(\bar{t_n}).$$

The rule is guaranteed to be monotonically increasing if for every derived literal $p_i(\bar{t_i})$, the arithmetic expression $\phi(p(\bar{t})) - \phi(p_i(\bar{t_i}))$ is always ≥ 0. This can be tested using symbolic manipulation on each expression $\phi(p(\bar{t})) - \phi(p_i(\bar{t_i}))$.

Condition Monotonicity_D: Consider an evaluation of a program P. Let p be a predicate defined in P, and S be the set of all predicates in P that are derived with respect to p (note that $p \in S$). Let $\mathcal{R}$ be the set of all the rules of P defining the predicates in S. The predicate p satisfies Condition Monotonicity_D iff every rule in $\mathcal{R}$ is monotonically increasing. □

Definition 4.3 Min-head-gap bounding function : For a predicate p as in Condition Monotonicity_D, a function γ mapping facts to integers is said to be a *min-head-gap bounding function for* p iff for each instance R' of any rule

R defining p, if $p(\overline{a})$ is the head fact and $q(\overline{b})$ is a derived fact in the body of R', $(\phi(p(\overline{a})) - \phi(q(\overline{b}))) \geq \gamma(q(\overline{b}))$. □

Note that the constant function $\gamma = 0$ is always a min-head-gap bounding function—however, one might be able to get a "better" function for the purposes of the subsequent theorem.

We can algorithmically determine a min-head-gap bounding function as follows. Suppose for each rule R defining p and for each derived predicate $p_i(\overline{t_i})$ in the body of R, each expression $\phi(p(\overline{t})) - \phi(p_i(\overline{t_i}))$ not only is non-negative but also (after simplification) has as arguments only variables from $\overline{t_i}$. Then we can derive a min-head-gap bounding function for p by symbolic arithmetic manipulations on these functions. The simplification above could include replacement of variables using arithmetic equalities present in the body of the rule. For instance, if we have the rule

$$fac(X, X * N) \leftarrow X > 0, Y = X - 1, fac(Y, N).$$

we can replace Y by $X - 1$ to get the min-head-gap bounding function (the constant function 1) for fac. See the full version of the paper for more details.

Theorem 4.1 *Consider a semi-naive evaluation where predicate p satisfies Condition Monotonicity_D and γ is a min-head-gap bounding function for p. Let S and $\mathcal{R}$ be as in Condition Monotonicity_D. In this evaluation, let F be the set of all the facts that have been derived for predicates defined in S, and $F' \subseteq F$ be a set of facts such that F' is locally saturated with respect to the set of rules $\mathcal{R}$. Let*

$$m = \min\{\phi(f) + \gamma(f) \mid f \in F - F'\}$$

If a fact $p(\overline{a})$ is such that $\phi(p(\overline{a})) < m$, then $p(\overline{a})$ will not be derived again. □

An analogous theorem holds with monotonically decreasing rules in place of monotonically increasing rules in Condition Monotonicity_D. The theorem gives us a way of ensuring Condition D for facts when the conditions on monotonicity are satisfied.

In an iteration of Basic Semi-Naive evaluation of an SCC, the set of facts in the p relations (i.e. the facts derived two or more iterations before) constitutes F' (as mentioned earlier) and the set of facts in the δp relations (i.e. those derived in the previous iteration) and the facts derived during the current iteration constitutes $F - F'$.

Note that although the set of derived predicates as well as the set of locally saturated facts depends on the actual evaluation used, the theorem holds independent of the evaluation.

Example 4.1 Consider the following program that computes a list of factorials of all squares of integers less than some constant n.

$$\begin{array}{ll} R1: fac\_list(0,[1]). & \\ R2: fac\_list(N,[V \mid L]) & \leftarrow N > 0, N < n, fac\_list(\\ & \quad N-1, L), fac(N * N, V) \\ R3: fac(0,1). & \\ R4: fac(N, N * V) & \leftarrow N > 0, N < n * n, \\ & \quad fac(N-1, V). \end{array}$$

Let the ϕ function map $fac\_list(N, \_)$ to N, and $fac(N, \_)$ also to N. We deduce that rule $R3$ and $R4$ are monotonically increasing. In rule $R2$, fac is a predicate from a lower SCC and hence is not derived wrt $R2$. Hence we deduce that $R1$ and $R2$ are monotonically increasing. Thus Condition Monotonicity_D is satisfied by predicates fac as well as fac_list. We also deduce min-head-gap bounding functions: the constant function 1 for fac as well as for fac_list.

From Theorem 4.1 we deduce that once a fac fact with index n is derived, no fac fact with index less than $n + 1$ will ever be derived again. We deduce similar results for fac_list. □

4.2 Monotonicity and Condition U

In this section we discuss how to use monotonicity of rules to satisfy Condition U. We make use of the definitions and results in Section 4.1. Let ϕ be a function as described earlier in this section.

Definition 4.4 Body_gap : Let R be a rule and let p' and q' be predicate occurrences in its body. Let R' be an instance of R with facts $p(\overline{a_1})$ and $q(\overline{a_2})$ used in the occurrences p' and q' respectively. We then define $body\_gap(R', p', q') = \phi(p(\overline{a_1})) - \phi(q(\overline{a_2}))$. If R has at least one derived predicate occurrence in its body, we define $body\_gap(R', q') = \max\{body\_gap(R', p', q') \mid p'$ is a derived predicate occurrence in $R\}$. If R has no derived predicate occurrence in its body, $body\_gap(R', q') = \infty$. □

Note that if there is only one derived predicate occurrence q' in the body of a rule R, and R' is any instance of R, then $body\_gap(R', q') = 0$.

Monotonicity can be used to infer that a fact can no longer be used in a body predicate occurrence q' based on Condition Monotonicity_U and Theorem 4.2 below.

Condition Monotonicity_U: Consider an evaluation of a program P. Let R be a rule with a body predicate occurrence q'. Let $p'_1, \ldots, p'_n$ be the derived predicate occurrences in the body of the rule R. Let γ be a function that maps q facts to integers. The predicate occurrence q' in R satisfies Condition Monotonicity_U with function γ iff, for each instance R' (with $q(\overline{a})$ used in the occurrence q')

$$body\_gap(R', q') \leq \gamma(q(\overline{a})) \qquad \square$$

Intuitively the theorem states that if two facts are used in a rule to make a successful derivation, the indices of the facts

are fairly "close" to each other. The function γ provides an upper bound on the gap. Suppose for each derived predicate occurrence p'_i in the body of rule R, $\phi(p'_i) - \phi(q')$ (after simplification) involves only the variables in the literal q'. Then, by a process similar to the derivation of min-head-gap bounding functions in Section 4.1, we can derive a function γ as in Condition Monotonicity_U.

Theorem 4.2 *Consider a semi-naive evaluation of a program P. Let R be a rule in P and q' be a body predicate occurrence in R such that q' satisfies Condition Monotonicity_U with function γ. Let m be an integer such that no fact for any $p'_i, 1 \leq i \leq n$ with index (under the function ϕ) less than m will be derived again*[6]*. Suppose that the set of all facts $\{p_i(\bar{b}) \mid 1 \leq i \leq n$ and $\phi(p_i(\bar{b})) < m\}$ is locally saturated with respect to R.*

Then, a fact $q(\overline{a_1})$ can no longer be used in the predicate occurrence q' of R if $\phi(q(\overline{a_1})) + \gamma(q(\overline{a_1})) < m$. □

The essential idea behind the theorem is as follows. Consider the index $\gamma(q(\bar{a})) + \phi(q(\bar{a}))$ for a fact $q(\bar{a})$. Suppose at a point in the evaluation we know that no new facts below this index will be derived, and all the facts below this index are locally saturated, then we know that every new rule instantiation must use in its body at least one fact above this index. But by Condition Monotonicity_U, for the fact $q(\bar{a})$ to be used all the derived facts in the rule body must have an index below $\gamma(q(\bar{a})) + \phi(q(\bar{a}))$. Hence we know that $q(\bar{a})$ cannot be used in any new rule instantiation beyond this point in the evaluation.

Note that the theorem makes no mention of whether q is derived with respect to the head of the rule or not. An analogous theorem holds when the *body_gap* of the rule with respect to q' is bounded from below, and no fact for any p'_i with index (under ϕ) greater than some m will be derived again.

A special case of the function γ is the constant function k (for some k). The above theorem generalizes the conditions of Sliding Window Tabulation ([NR90]), since only such constant functions could be used for γ in Sliding Window Tabulation. Example 4.2 shows the need for allowing general functions.

Example 4.2 We use the program from Example 4.1 again. Consider Rule $R4$:

$$R4 : fac(N, N * V) \leftarrow N > 0, N < n * n, fac(N - 1, V).$$

There is only one derived predicate in the body of this rule, hence a fac fact can be used at most once (Condition Bounds_U). Another way of looking at this is using monotonicity. A γ function on fac that bounds *body_gap* is the constant function 0. Hence if no fac fact with index less than n will be derived henceforth, fac facts with indices less than n will no longer be used in this rule. A similar result holds for uses of fac_list facts in rule $R2$ shown below:

$$R2 : fac\_list(N, [V \mid L]) \leftarrow N > 0, N < n, fac\_list(N - 1, L), fac(N * N, V).$$

The one predicate occurrence left is the occurrence of fac in rule $R2$. Now we derive a function γ on fac facts that satisfies Condition Monotonicity_U, using the technique described earlier: γ maps $fac(N * N, \_)$ to $N - 1 - N * N$, and hence $fac(M, \_)$ to $\sqrt{M} - M - 1$. Using this we deduce that if no fac_list facts with index less than n will be produced and there are no fac_list facts with index less than n in the differential δ relations, then fac facts $fac(M, \_)$ such that $M + \sqrt{M} - M - 1 < n$ will no longer be used. But from Example 4.1 we know how to find what fac_list facts will no longer be produced: if a fact $fac\_list(n, \_)$ has been produced in an iteration, no fac_list fact with index less than $n + 1$ will be produced hence.

Thus in a semi-naive evaluation, one iteration after $fac\_list(n, \_)$ has been produced we know that any $fac(m, \_)$ fact with $\sqrt{m} - 1 < n$ can no longer be used in the occurrence of fac in rule $R2$. □

5 Synchronization

A synchronizing technique orders the computation of a program so that derivations of facts are "close" to their uses; this helps reduce the "life-spans" of facts. Intuitively, if each fact is stored for only a short time during the computation, the total space required for the overall computation is reduced. Here we only discuss one of the three techniques presented in the full version of this paper ([SSRN90]) due to lack of space.

Interleaved-SCC synchronization is a form of synchronization that exploits SCC structure. The intuition behind the technique is as follows. Consider a predicate p defined in an SCC. A p-fact must be retained until Conditions U and D are satisfied by it in this ("producer") SCC; in addition, it must be retained until it has been used completely in all occurrences of p in other ("consumer") SCCs. If our evaluation proceeds SCC-by-SCC, the producer SCC computation must be completed before computation of the consumer SCCs can begin, and p-facts must therefore be retained at least until the end of the computation of the producer SCC. However, it is sometimes possible to use the p fact in all consumer SCCs soon after it is produced by interleaving the execution of SCCs, thereby making it possible to discard it sooner, while retaining all the advantages of an SCC by SCC semi-naive evaluation. We present our techniques for the case of one producer SCC, one consumer SCC, and one predicate defined in the producer and used by the consumer. In the full version of the paper we present the general form of our techniques which removes these assumptions.

Let a predicate p be defined in an SCC $S1$ and used in an

[6] Theorem 4.1 may be used to ensure this.

SCC $S2$. SCC $S1$ is referred to as the producer and $S2$ as the consumer.

Condition Interleaved-SCCs:

- The producer and the consumer SCCs contain only monotonically increasing rules.
- In the consumer $S2$, suppose a rule R contains an occurrence of p in the body. Call this predicate occurrence p'. Then either
  (1) There is a bound $max_{p'}$ such that for any fact $p(\bar{b})$ that can be used in the occurrence p', $\phi(p(\bar{b})) \leq max_{p'}$
  or
  (2) Suppose q is any derived predicate and q' any occurrence of q in the body of R. Then there must exist a function $\gamma_{p',q'}$ that maps q facts to integers such that for each instance R' of R (where say $q(\bar{b})$ is used in the occurrence q'), $body\_gap(R', p', q') \leq \gamma_{p',q'}(q(\bar{b}))$. □

We can derive bounding functions $\gamma_{p',q'}$ in a manner similar to to the way we derive bounding functions for Condition Monotonicity_U in Section 4.2.

We now describe the *Interleaved-SCC* synchronization technique, which works on any subprogram that satisfies Condition Interleaved-SCCs. At each stage of the evaluation we know what facts have been newly derived. During the evaluation we can therefore easily keep track of the maximum ϕ values for facts of each derived predicate in each of the SCCs. From this value and the functions $\gamma_{p',q'}$ we can deduce that with the given derived facts only p facts with an index less than a certain value can be used. We can easily derive this index (we do not describe here the straightforward way to compute this index), and we call this index max_p in the following algorithm:

Procedure Interleaved-SCC_Producer $(S1, max\_p)$
(1) Evaluate $S1$ till no facts $p(\bar{b})$ such that $\phi(p(\bar{b})) \leq max\_p$ can be derived.
/* Tested using monotonicity of rules in $S1$ */
/* Facts in $S1$ can be discarded once they satisfy Conditions D and U */
end Interleaved-SCC_Producer

Procedure Interleaved-SCC_Consumer $(S2)$
(1) Evaluate $S2$ with the following restriction:
(2) Whenever new facts are made available for derived predicates in $S2$, before using the facts do
(3) Update the index max_p.
(4) Call Interleaved-SCC_Producer $(S1, max\_p)$.
end Interleaved-SCC_Consumer

Example 5.1 Consider again the fac_list program from Example 4.1. This program has two SCCs, the lower one containing the predicate fac and the upper one containing fac_list. We call the lower SCC $S1$ and the upper one $S2$. SCC $S1$ is a producer of fac and $S2$ a consumer.

There is only one rule $R2$ in $S2$ that uses the predicate fac. This rule has a derived predicate fac_list. We derive the function γ that maps $fac\_list(N-1, \_)$ to $N^2 - N + 1$, (and hence $fac\_list(N, \_)$ to $N^2 + N + 1$) to bound $body\_gap(R2, fac(N * N, V), fac\_list(N-1, L))$. SCCs $S1$ and $S2$ satisfy Condition Interleaved-SCCs with this function γ that bounds body-gap. We can then use Interleaved_SCC evaluation to evaluate this program.

After each semi-naive iteration of the higher SCC (in Procedure Interleaved_SCC_Consumer) new facts are produced. Using these facts we find the maximum value of $\phi(fac\_list(N, \_)) + \gamma(fac\_list(N, \_))$. Thus if $fac\_list(n, \_)$ has been produced, we need fac facts with indices up to $n^2 + 2n + 1$. We then call Procedure Interleaved_SCC_Producer$(S1, n^2 + 2n + 1)$. SCC $S1$ then iterates, producing fac facts. Due to monotonicity of rules in $S1$, we know that when $fac(n^2 + 2n + 1, \_)$ has been produced, all fac facts with indices $\leq n^2 + 2n + 1$ have been produced. Hence Procedure Interleaved_SCC_Producer returns, and Procedure Interleaved_SCC_Consumer continues with its next iteration.

We discard each fact once it satisfies Condition D and U (see Examples 4.1 and 4.2). It is easy to see that in SCC $S2$, only two fac_list facts are retained at any time; each fac_list fact uses $O(n)$ space. As for SCC $S1$, we store at most facts with indices from $(n-1)^2$ to n^2, which means at most $2n-1$ facts are stored. Thus we use a total of $O(n)$ space using this space optimization technique. By discarding facts during the evaluation, we have improved the space utilization from $O(n^2)$ to $O(n)$. □

5.1 Using Inverted Rules

In several cases (such as monotonically increasing SCCs that have been rewritten using the Magic Sets transformation), the conditions for Interleaved SCCs are almost met, except that the two SCCs are monotone in opposite directions. By using the notion of inverted rules, we can still use Interleaved-SCC evaluation in some cases. Consider the following example:

Example 5.2 The following Magic rewritten program P^{mg}_{Fac} is used to compute the nth Factorial number.

$R1: m\_fac(n).$
$R2: fac(0,1) \leftarrow m\_fac(0).$
$R3: m\_fac(N-1) \leftarrow m\_fac(N), N > 0.$
$R4: fac(N, N * X1) \leftarrow m\_fac(N), N > 0, fac(N-1, X1).$

There are two SCCs in this program—SCC $S1$ defining m_fac and SCC $S2$ defining fac. Unfortunately, the conditions for Interleaved-SCC synchronization are not satisfied, since the two SCCs are monotone in opposite directions. Each $m\_fac(i)$ fact is used in the computation of

$m\_fac(i-1)$ (using rule R3) and in the computation of $fac(i)$ (using rule R4). Since none of the rules defining fac can be applied until all the m_fac facts have been computed, the two uses of an m_fac fact are considerably "separated" in the evaluation.

If an m_fac fact is not discarded until it has been used to compute the corresponding fac fact, we do not achieve any savings in the space complexity of this program; we still need to store $O(n)$ facts. □

Naughton and Ramakrishnan [NR90] introduced the notion of inverted magic rules. Suppose a set of rules is monotone. The set of *fringe facts* for these rules are those that do not generate any new facts. We can in some cases use the original set of rules in reverse—feed them the head facts and regenerate the body facts. This is done using "inverted" rules created by swapping the head and one of the body literals in a rule. A set of rules $\mathcal{R}$ is said to be *invertible* if such a set of inverted rules can be created from $\mathcal{R}$.

Condition Inverted_Rules: A pair of SCCs S_0, S_1 satisfies this condition if:

- These SCCs satisfy the conditions for Interleaved-SCC synchronization with S_0 as the producer SCC and S_1 as consumer SCCs, except that the rules in S_0 are monotonic in the opposite direction to the rules in S_1.
- The set $\mathcal{R}_0$ of rules in S_0 is invertible. □

Suppose S_0 and S_1 satisfy Condition Inverted_Rules. Now, the inverted rules $\mathcal{R}_0'$ obtained from $\mathcal{R}_0$ satisfy Condition Interleaved-SCCs with $\mathcal{R}_0'$ as the producer SCC and S_1 as the consumer SCC.

If a given set of rules $\mathcal{R}$ is invertible, the inverted rules $\mathcal{R}'$ provide a mechanism to ensure that any (non-fringe) fact computed by an evaluation of $\mathcal{R}$ can be discarded without violating Condition U—these facts can be recomputed using $\mathcal{R}'$ and the set of fringe facts. If Condition Inverted_Rules is satisfied for a pair of SCCs, we can use the following variant of Interleaved-SCC synchronization:

Algorithm Inverted_Rules_Eval (S_0, S_1)

Let the set of inverted rules obtained from $\mathcal{R}_0$ be $\mathcal{R}_0'$.

(1) Evaluate S_0. Discard facts other than fringe facts once they are no longer needed for use in S_0 (based on Conditions D and U restricted to S_0).

(2) Evaluate the set of rules $\mathcal{R}_0'$ (with the fringe facts) and S_1 using Interleaved-SCCs synchronization (with $\mathcal{R}_0'$ as the producer SCC).

end Inverted_Rules_Eval

This potentially reduces the number of facts that need to be stored at any point in the evaluation, as the following example illustrates. In the full version of the paper, we generalize the notion of inverted rules beyond that of [NR90], and allow multiple SCCs to use facts generated by the inverted rules.

Example 5.3 We continue with Example 5.2. Condition Inverted_Rules is satisfied by the pair of SCCs. The pair of SCCs are monotone in opposite directions, but satisfy Condition Interleaved-SCCs otherwise. A γ function which is the constant function 1 is used for rule $R4$. For rule $R2$ we have a bound 0 such that any m_fac fact used here has index ≤ 0. The rules in the producer SCC $S1$ can be inverted. There is only one inverted rule which is as follows[7]:

$$R' : m\_fac(N+1) \leftarrow m\_fac(N), N < n.$$

In the first phase of Algorithm Inverted_Rules_Eval we evaluate $S1$. Facts other than fringe facts can be discarded during the evaluation of $S1$ once they satisfy Conditions D and U with respect to $S1$ alone. We can use either monotonicity or bounds on number of uses of facts to satisfy Conditions D and U. In either case, we store only two facts during this evaluation. A fact $m\_fac(i)$ is discarded once it has been used to compute $m\_fac(i-1)$; since however $m\_fac(i)$ is needed in the computation of $fac(i)$, it is recomputed in the second phase of Algorithm Inverted_Rules_Eval using the inverted magic rule R'.

In the second phase we use Interleaved-SCC synchronization with the producer being rule R' and the consumer being $S2$. We skip the details here, but note that at most two m_fac facts and two fac facts are stored at a time. We have thus reduced the space complexity from $O(n)$ to $O(1)$. Sliding Window Tabulation is applicable on this program, and would also use $O(1)$ space.

Sliding Window Tabulation is not applicable, however, on the magic program obtained from the fac_list program (described in previous sections), whereas Inverted_Rules_Eval (with the extension to multiple consumer SCCs described in the full paper) is applicable. □

6 Combining Techniques

The identification of the three components of any space optimization technique helps us to mix and match different techniques for different parts of a program.

Our approach to combining techniques consists of first organizing the program into subprograms and choosing synchronization strategies for each subprogram. We first identify certain subprograms that are to be synchronized using Nested-SCC synchronization, based upon how the rewriting changes the SCC structure of the original program. We then try to use Interleaving to further reduce space needs by examining producer-consumer subprograms. Next, we examine each subprogram, and consider how delaying first

[7] The inverted magic rule generated in [NR90] did not have the condition $N < n$ in it, but the evaluation ensured that computation did not proceed beyond n.

use of facts can be used to improve space utilization. This completes the choice of synchronization strategies.

We then check which techniques for ensuring Conditions U and D are applicable, given the choice of synchronization strategies and choose techniques for each part of the program. This completes the decisions on space optimization. A detailed algorithm may be found in the full version of the paper.

7 Overheads

There are three aspects to the overheads involved with our space optimization techniques.

Compile-time time overheads: Suppose we are given a) dependency information about all predicates in the program, b) duplicate-freedom information, c) ϕ functions for all predicates in the program, and d) γ functions for different predicates as necessary. Then the cost of testing various conditions is linear in the size of the input. We have indicated briefly how to derive some of the functions, and we expect our algorithms to be efficient in practice.

Run-time time overheads: These overheads are minimal for tests based on bounds—in some cases there is no overhead for any tests, and at most, in other cases, a few simple counts need to be maintained for each fact, and updated when the fact is used. Tests based on monotonicity are a little more complicated. When a fact is derived we need to compute its ϕ value, and possibly its value under some of the γ functions. This computation is constant time per fact and quite efficient. The only important cost here is the cost of secondary indices on the ϕ value so that facts can be discarded when index m (from Theorem 4.1) reaches a certain value.

Run-time space overheads: For bounds based techniques, there is no overhead in some cases, and a constant overhead of one to a few integers per stored fact in other cases. For monotonicity based techniques, we can choose to either store various function values for each stored fact, or recompute them on demand and thus avoid all space overheads. There is at most a constant space overhead per stored fact, even if we decide to store the function values. When the number of facts stored is reduced by an order of magnitude, a constant space overhead per stored fact is clearly negligible.

8 Conclusion

In this paper we have described how to reduce the space required during bottom-up evaluation of logic programs by discarding facts. We showed that any space optimization technique that discards facts during the evaluation has three basic components: (1) ensuring that all derivations are made, (2) ensuring that derivations are not repeated, and (3) synchronizing the derivation and use of facts. We presented some techniques for ensuring each of these three components, and showed how they can be combined to get a space optimization technique for the full logic program. Since Sliding Window Tabulation [NR90] can be shown to be just one way of combining techniques for each of these three components, our results subsume those in [NR90].

Incorporating these optimizations into an actual deductive database runtime system, and enhancing the compiler/interpreter to test for and use these optimizations is being considered for CORAL, a deductive database system being developed at the University of Wisconsin, Madison.

Future work in this area includes finding more methods for ensuring each of the three components of an effective space optimization technique. For instance, the generate and test paradigm could benefit from a form of synchronization where facts are generated and tested in a synchronized fashion, and may be discarded once they have been tested. Work is also needed in determining which technique to use when more than one technique is applicable to a given part of the program.

References

[BR87] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.

[Hir75] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[MR90] Michael J. Maher and Raghu Ramakrishnan. Dèjá vu in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming*, Cleveland, Ohio, 1990.

[NR90] Jeffrey F. Naughton and Raghu Ramakrishnan. How to forget the past without repeating it. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.

[RSS90] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.

[SSRN90] S. Sudarshan, Divesh Srivastava, Raghu Ramakrishnan, and Jeff Naughton. Space optimization in the bottom-up evaluation of logic programs. Technical Report Tech. Report (To appear), Univ. Wisconsin at Madison, Madison WI 53706, 1990.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.