Space Optimization in Deductive Databases

DIVESH SRIVASTAVA
AT & T Bell Laboratories

S. SUDARSHAN
Indian Institute of Technology
and
RAGHU RAMAKRISHNAN and JEFFREY F. NAUGHTON
University of Wisconsin

In the bottom-up evaluation of logic programs and recursively defined views on databases, all generated facts are usually assumed to be stored until the end of the evaluation. Discarding facts during the evaluation, however, can considerably improve the efficiency of the evaluation: the space needed to evaluate the program, the I/O costs, the costs of maintaining and accessing indices, and the cost of eliminating duplicates may all be reduced. 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. We show that every such space optimization method has certain components, the first to ensure soundness and completeness, the second to avoid redundancy (i.e., repetition of derivations), and the third to reduce "fact lifetimes" (i.e., the time period for which each fact must be retained during evaluation). We present new techniques based on providing bounds on the number of derivations and uses of facts, and using monotonicity constraints for each of the first two components, and provide novel synchronization techniques for the third component of a space optimization method. We describe how techniques for each of the three components can be combined in practice to obtain a space optimization method for a program. Our results are also of importance in applications such as sequence querying, and in active databases where triggers are defined over multiple "events."

A preliminary version of this paper appeared in the Proceedings of the ACM SIGMOD Conference on the Management of Data, Denver, Colorado, 1991.
Authors' addresses: D. Srivastava, AT & T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974; email: ⟨divesh@research.att.com⟩; S. Sudarshan, Department of Computer Science and Engineering, Indian Institute of Technology, Powai, Bombay 400 076, India; email: ⟨sudarsha@cse.iitb.ernet.in⟩; R. Ramakrishnan and J. F. Naughton, Computer Sciences Department, University of Wisconsin, Madison, WI 53706; email: ⟨raghu@cs.wisc.edu⟩; ⟨naughton@cs.wisc.edu⟩.
The work of the first three authors was supported in part by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award, with matching grants from Digital Equipment Corporation, Tandem, and Xerox, NSF grants IRI-8804319 and IRI-9011563, and an IBM Faculty Development Award. The work was performed while the first two authors were at the University of Wisconsin, Madison. The work of J. F. Naughton was supported by NSF grant IRI-8909795.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*query languages*; H.2.4 [**Database Management**]: Systems—*query processing*; H.2.m [**Database Management**]: Miscellaneous—*deductive databases*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Bottom-up query evaluation, deductive database systems, discarding facts, logic programming

1. INTRODUCTION

Bottom-up evaluation of a query on a recursively defined view/logic program proceeds by repeatedly applying program rules to generate facts until no new facts can be computed. Bottom-up evaluation has been shown to have several advantages over Prolog-style top-down evaluation in the area of deductive databases (see, for example, [Ullman 1989]). The primary advantages are: (a) the evaluation is set-oriented and can benefit from efficient join techniques; (b) repeated computation is avoided as a result of storing answers to subqueries and reusing them; and (c) cyclic subqueries are detected and the resulting infinite loops are avoided, making the evaluation not only terminating, but also complete, in many cases where Prolog-style top-down evaluation would loop.

However, a disadvantage of bottom-up evaluation is that all generated facts are usually assumed to be stored until the end of the evaluation. Because 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. 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, and the like are also reduced. Thus, discarding facts during the evaluation can result in time as well as space improvements. We refer to evaluation methods that discard facts during the course of the evaluation of a logic program (instead of just at the end of the evaluation) as *space optimization methods.*[1]

Naughton and Ramakrishnan [1994] introduced the subject of space optimization methods in database program evaluation, and presented one method, Sliding Window Tabulation, that reduces the space utilized during the evaluation of a restricted class of programs rewritten using Magic Sets [Beeri and Ramakrishnan 1991].

Our paper provides a general framework for, and identifies the key components of, space optimization methods. Prematurely discarding a fact could mean that some derivation that uses this fact may not be made; this could

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

affect the set of answers to the user's query. Intuitively, we have the following condition for correctness of any space optimization method (the condition is formalized later):

Soundness and Completeness. For each generated fact, it must be ensured that all facts that can be derived using it are actually derived. Further, no fact should be derived that would not be derived without space optimization.[2]

Even if soundness and completeness are ensured, repeated derivations of a discarded fact may not be recognized as yielding duplicates; this could lead to repeated inferences using this fact. Intuitively, it is desirable to also ensure the following:

Nonredundancy. Repeated occurrences of the same derivation must be avoided.

Given techniques for discarding facts while ensuring soundness, completeness, and nonredundancy, we can readily use them in conjunction with any evaluation method, for example, semi-naive fixpoint evaluation. However, the opportunities for discarding facts can often be increased significantly by modifying the order in which the derivations are made in a program evaluation in order to ensure that derivations of facts are "close" to all their uses; this could be used to reduce the necessary "lifetime" of the fact. Thus a third aspect of any space optimization method is:

Reducing Fact Lifetimes. It is desirable to make derivations of facts be "close" to all their uses.

Indeed, every space optimization method has certain components, the first to ensure soundness and completeness, the second to avoid redundancy, and the third to reduce fact lifetimes. 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 methods.

In this paper, we present several novel techniques and program evaluation strategies for ensuring the soundness, completeness, and nonredundancy requirements, and for reducing fact lifetimes. We also discuss how to mix and match various techniques useful for different parts of a program to get a space optimization method for the full program in a modular fashion. This has the following important benefits:

(1) We obtain a much deeper understanding of how space optimization can be achieved.
(2) Concretely, by instantiating our "mix-and-match" algorithm with specific techniques and evaluation strategies corresponding to each of the preceding three components, we can optimize a much larger class of programs than the Sliding Window Tabulation method of Naughton and Ramakrishnan [1994] (see Examples 1.2.1 and 8.3.1). In fact, we show

[2]This point is relevant if nonmonotonic constructs such as negation and aggregation are present.

that Sliding Window Tabulation is just one particular way of combining techniques for each of the three components.

1.1 Applications

Space optimization is important, in general, for query evaluation in deductive databases, as the examples later in the paper illustrate.

An application area where space optimization methods are particularly important is sequence querying (e.g., Roth et al. [1993] and Seshadri et al. [1994]). The longest common subsequence problem (Example 1.2.2) is representative of many problems that arise in DNA sequence analysis. Example 8.3.1 considers the problem of computing N-day averages, which is representative of many problems that arise in querying stock market sequence data (see, e.g., Roth et al. [1993]). In each of these cases, space-efficient evaluation techniques enable the answering of queries on larger sequence databases than were feasible without the use of our space optimization techniques.

Space optimization methods can also be used in active databases. Active database models, such as Compose [Gehani et al. 1992], permit users to specify patterns of events (event expressions) that trigger specific actions. Compose only allows regular expressions as patterns to ensure constant additional storage requirements. However, the need for more complex patterns, in particular, dealing with time and sequences, soon became evident [Jagadish et al. 1992]. Space optimization is very important to bound the space utilized for detecting such complex patterns. Such patterns can be expressed using logic programs, and our techniques (in particular the monotonicity-based techniques described in Section 6) are then directly applicable. Our techniques also can be extended to deal directly with the syntax used for specifying such complex events, avoiding the need to translate the patterns into logic programs.

1.2 Motivating Examples

We present some examples that underscore the importance of space optimization in deductive databases.

Example 1.2.1. Consider the problem of computing the ancestors of a given person, an important problem in deductive database literature. We are given a binary relation $father(X, Y)$, with the intended meaning that Y is the father of X. Then the following program defines the relation $anc(X, Y)$, with the intended meaning that Y is an ancestor of X.

$$anc(X, Y) :- father(X, Y).$$
$$anc(X, Y) :- father(X, Z), anc(Z, Y).$$
$$\text{Query: } ?\text{-}anc(n, X).$$

Suppose that the *father* relation is an acyclic relation, with the functional dependency *father*: \$1 → \$2, that is, each person has at most one father.

Consider a database with the following facts in the *father* relation: $father(n, n-1), father(n-1, n-2), \ldots, father(2, 1), father(1, 0)$.

If we take the bottom-up approach of rewriting by Magic Sets [Ramakrishnan 1988] followed by semi-naive bottom-up evaluation [Bancilhon 1985], the space and time required is $O(n^2)$. This is because the evaluation computes the following $O(n^2)$ *anc* facts: $anc(n, n-1), \ldots, anc(n, 1)$, $anc(n, 0)$, $anc(n-1, n-2), \ldots, anc(n-1, 1)$, $anc(n-1, 0), \ldots, anc(1, 0)$.

Sliding Window Tabulation, as described in Naughton and Ramakrishnan [1994], is not applicable to this program because the program does not exhibit any monotonicity. The techniques described in this paper can reduce the space required to answer the query to $O(n)$. [However, the asymptotic time complexity remains $O(n^2)$.] The intuition is as follows:

—In an SCC-by-SCC bottom-up evaluation of the Magic Sets rewritten program, each *anc* fact can be used *only* in the iteration subsequent to the iteration in which it is derived. (This can be deduced from the fact that each of the program rules is linear.)

—The program is "duplicate-free"; that is, no *anc* fact is derived using two different rule instances. (This can be deduced from the acyclicity of the *father* relation, along with the functional dependency between the arguments of the *father* relation. The actual facts in the *father* relation are not needed to infer the duplicate-freedom property.)

Hence, we can discard those *anc* facts that are not answers to the query one iteration after they are computed. This results in an $O(n)$ space complexity.[3] Retaining all the computed *anc* facts until the end of the evaluation results in an $O(n^2)$ space complexity.

The following example is from Naughton and Ramakrishnan [1994]. This is a complex program, and can be omitted in a first reading without loss of continuity.

Example 1.2.2. Consider the problem of computing the length of the longest common subsequence (LCS) of two strings a and b. This problem is significant because it is representative of a number of problems that arise in DNA sequence analysis, an area that has been identified as a promising application for deductive database technology (e.g., Tsur et al. [1990]).

We are given two strings, say $A = a_0 a_1 \cdots a_{m-1}$ and $B = b_0 b_1 \cdots b_{n-1}$, where the a_i and b_j are drawn from some common alphabet. We use the standard algorithm of Hirschberg [1975] to compute the LCS of two strings.

[3]The program "factoring" transformations described in Naughton et al. [1989] and Kemp et al. [1990] also result in an $O(n)$ space complexity for evaluating this query, although no facts need be discarded before the end of the evaluation. If space optimization is applied to the "factored" program, and answers are returned to the user as they are computed, then the query can be evaluated in constant space!

To express the problem in deductive database notation, 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. $b(j, \alpha)$). Then the following program defines the relation $lcs(M, N, X)$, with the intended meaning that the longest common subsequence of A beginning at a_M and B beginning at b_N is of length X.

$lcs(m, N, 0).$
$lcs(M, n, 0).$
$lcs(M, N, X) :- M < m, N < n, a(M, C), b(N, C), lcs(M+1, N+1, X-1).$
$lcs(M, N, X) :- M < m, N < n, a(M, C), b(N, D), C \neq D, lcs(M+1, N, X1),$
$lcs(M, N+1, X2), X = \max(X1, X2).$
Query: ?-$lcs(0, 0, X)$.

If the strings are of length m and n, then evaluating the program using the top-down Prolog evaluation strategy gives a running time that is $\Omega(\binom{m+n}{n})$. The function $\binom{m+n}{n}$ grows extremely quickly. For example, if $m = n = 20$, we have $\binom{m+n}{n} > 275 \times 10^9$; if $m = n = 100$, we have $\binom{m+n}{n} > 1.8 \times 10^{59}$. Clearly, the Prolog evaluation strategy cannot be used on this program for any but the shortest of strings.

If we take the bottom-up approach of rewriting by Magic Sets [Beeri and Ramakrishnan 1991] followed by semi-naive bottom-up evaluation, the running time is reduced to $O(mn)$. This is a dramatic improvement; unfortunately, the space required is also $O(mn)$. In DNA sequence analysis, comparison of strings of over 10^4 bases will be routine. (The human genome is estimated to contain over 10^9 base pairs.) Even if each fact to be stored fits in a single byte, on strings this size, the standard bottom-up approach will require over a hundred megabytes (10^8 bytes) of storage.

Sliding Window Tabulation, as described in Naughton and Ramakrishnan [1994], evaluates this program in $O(m + n)$ space (which is just 10^4 bytes) and $O(mn)$ time, by discarding facts in the course of the evaluation. Thus this improvement in the space complexity is very important if the program is to be run over such large databases.

Sliding Window Tabulation is effective on the LCS example, but it does not work on many simple variations. 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 preceding program is embedded in a larger program that uses the length of the longest common subsequence to perform further analysis, such as finding the region of a given DNA sequence that best matches the given test sequence, Sliding Window Tabulation is again inapplicable. The techniques described in this paper can handle such extensions and are applicable to a much larger class of programs.

2. DEFINITIONS

In this paper, we consider Horn clause logic programs,[4] and assume the usual definitions including those of terms, atoms, and rules (clauses). We assume some familiarity with semi-naive evaluation and, in some of the sections of the paper, with the Magic Sets transformation. We refer the reader to Ullman [1989] for more details.

We present intuitive definitions of some well-known concepts in logic programming; see Lloyd [1987] for formal definitions. Informally, the universe of a program consists of all the values (such as integers, reals, and strings) that the program can manipulate. A *substitution* σ is a mapping from a set of variables to values in the universe of the program; it is extended to handle syntactic objects (such as terms/literals) containing variables in a straightforward manner. For example, $\sigma = \{X/4, Y/\text{"}john\text{"}\}$ is a substitution, and $p(X,Y)[\sigma]$ (the result of applying σ to $p(X,Y)$) is $p(4,\text{"}john\text{"})$. The result of applying a substitution to a syntactic object is called an *instance* of the object. In this paper, we only use substitutions that map *all* the variables in the input term to values in the universe (ground values).

A *program* is treated as a set of rules and database (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 the program. In this paper, we assume that all program facts are ground, that is, they do not contain any (universally quantified) variables. A sufficient condition that guarantees this is *range-restrictedness* of the program, that is, for each rule in the program, each variable that appears in the head of the rule also appears in the body of the rule.

2.1 Program Evaluations

The *meaning* of a program is defined as its least fixpoint in the Herbrand universe [Lloyd 1987]. The least fixpoint of a program can be computed iteratively, using a *bottom-up evaluation*. Bottom-up evaluation of a program computes all facts that can be inferred starting from the facts in the database, and using the program rules repeatedly.

In this paper we describe several bottom-up evaluation techniques for logic programs—each has some advantages and some disadvantages. However, we would like to make several claims that are applicable to each of these evaluation techniques and hence we need an abstract notion of an evaluation of a program. For this purpose, we first define program states and state transitions.

A *state* in a program evaluation is a pair $\langle \mathcal{F}, \mathcal{H} \rangle$, where $\mathcal{F}$ and $\mathcal{H}$ are disjoint sets of facts: intuitively, $\mathcal{F}$ denotes the facts that are *available* for use in derivations from this state, $\mathcal{H}$ denotes the facts that have been derived

[4]Our techniques can be extended to include some classes of programs with negation and aggregation; we do not do so for simplicity.

but are *hidden*, and hence not available for use in derivations from this state; hiding of facts is used in some of the evaluation strategies we describe later.

The body of a rule instance $R[\sigma]$ is *satisfied* in a state $\langle \mathcal{F}_i, \mathcal{H}_i \rangle$ if each positive literal in the body of $R[\sigma]$ is present in the set $\mathcal{F}_i$.[5] For example, if R is the rule:

$$q(X) :- p(X).$$

and the substitution σ is $\{X/a\}$, then the body of $R[\sigma]$ is satisfied in the state $\langle \{p(a)\}, \{p(b)\} \rangle$. However, if γ is $\{X/b\}$, then the body of $R[\gamma]$ is not satisfied in that state.

Definition 2.1.1 (*Derivation Step*). A *derivation step* at state $\langle \mathcal{F}_i, \mathcal{H}_i \rangle$ consists of a rule R along with a substitution σ on its variables, such that the body of $R[\sigma]$ is satisfied.

The head of $R[\sigma]$ is referred to as the fact *derived* in this step, and the instantiated body literals are referred to as the facts *used* in this derivation step.

We often use the term "derivation" to refer to a derivation step.

A *state transition* is defined as a mapping from one state to another $\langle \mathcal{F}_1, \mathcal{H}_1 \rangle \rightarrow^{\mathcal{S}} \langle \mathcal{F}_2, \mathcal{H}_2 \rangle$. $\mathcal{S}$ is the collection of derivation steps, possibly empty, associated with this transition. We require exactly one of the following conditions to hold in each state transition:

New Available Facts: $\mathcal{F}_2 = \mathcal{F}_1 \cup$ facts derived in $\mathcal{S}$, and $\mathcal{H}_2 = \mathcal{H}_1$.
New Hidden Facts: $\mathcal{H}_2 = \mathcal{H}_1 \cup$ facts derived in $\mathcal{S}$, and $\mathcal{F}_2 = \mathcal{F}_1$.
Show Hidden Facts: $\mathcal{S} = \phi$, $\mathcal{F}_2 = \mathcal{F}_1 \cup \mathcal{H}$, and $\mathcal{H}_2 = \mathcal{H}_1 - \mathcal{H}$, where $\mathcal{H} \subseteq \mathcal{H}_1$.
Discard Facts: $\mathcal{S} = \phi$, $\mathcal{F}_2 \subset \mathcal{F}_1$, and $\mathcal{H}_2 = \mathcal{H}_1$.

In the *initial state*, $\mathcal{F}$ is the set of EDB facts in the program, and $\mathcal{H}$ is empty (because no facts are hidden initially).

Definition 2.1.2 (*Evaluation*). Consider a program P. An *evaluation* of P is a sequence of state transitions starting from the initial state. Each of the states in this sequence is referred to as a *point in the evaluation* of P.

The notion of earlier and later points with respect to a given point in an evaluation is the natural one.

We discussed the idea of derivation steps earlier; in an actual bottom-up evaluation, a set of derivation steps using a rule are typically performed together in a "rule application." The following definition formalizes rule applications.

Definition 2.1.3 (*Rule Application*). An *application* of rule R in state $\langle \mathcal{F}_1, \mathcal{H}_1 \rangle$ is a transition $\langle \mathcal{F}_1, \mathcal{H}_1 \rangle \rightarrow^{\mathcal{S}} \langle \mathcal{F}_2, \mathcal{H}_2 \rangle$ such that all derivation steps in

[5] When negation is allowed, we also require that each instantiated negative literal is not present in $\mathcal{F}_i$, in addition to other requirements.

S are performed using rule R, and $\mathcal{F}_2 \cup \mathcal{H}_2$ contains all facts that can be derived using rule R in the state $\langle \mathcal{F}_1, \mathcal{H}_1 \rangle$.

An application of a *set of rules* $\mathcal{R}$ is a transition as shown, where all derivation steps in $\mathcal{S}$ are performed using rules $R \in \mathcal{R}$, and $\mathcal{F}_2 \cup \mathcal{H}_2$ contains all facts that can be derived using the rules $R \in \mathcal{R}$ in the state $\langle \mathcal{F}_1, \mathcal{H}_1 \rangle$.

Notice that the preceding definition does not require $\mathcal{S}$ to contain all the derivation steps that can be performed. Thus derivations that have been made earlier need not be repeated. Actually, if a fact is present in $\langle \mathcal{F}_1, \mathcal{H}_1 \rangle$, there is no need to perform a derivation step that derives the fact, regardless of whether the derivation step was performed earlier.

We often say "*apply rule R*" (resp. "*apply a set of rules* $\mathcal{R}$") in a given state to indicate that the state transition defined by the application of rule R (resp. $\mathcal{R}$) must be carried out. Specific details of how a rule application is carried out, such as the order of the join operations and the technique used for each join operation, are not relevant to the results in this paper.

Definition 2.1.4 (S-*Evaluation*). An S-*evaluation* is an evaluation that satisfies:

(1) Each state transition where new facts are derived is carried out by an application of a rule or a set of rules.
(2) In each state transition that discards facts, precisely one fact is discarded.

The motivation for the second requirement is discussed later.

In general, the evaluation of a program depends upon the *evaluation method* that is used. For simplicity, in this paper we only consider evaluation methods that produce a unique S-evaluation for each program and initial state. In the rest of this paper, an evaluation is implicitly assumed to be an S-evaluation unless otherwise stated.

2.2 Semi-Naive Evaluation

Definition 2.2.1 (*Semi-Naive Evaluation*). We say that an evaluation "has the *semi-naive property*" if (a) no derivation step occurs more than once in any transition, and (b) no two transitions in the evaluation contain the same derivation step. We call such an evaluation a *semi-naive evaluation*.

An evaluation is said to be *semi-naive with respect to a predicate p* if the preceding two conditions hold for all derivation steps that derive p facts.

Note that an evaluation is semi-naive if and only if it is semi-naive with respect to each of the predicates defined in the program.

Several techniques for evaluating programs in a semi-naive fashion have been proposed, for example, Bancilhon [1985], Balbin and Ramamohanarao [1987], and Ramakrishnan et al. [1994]. We briefly outline the semi-naive evaluation method presented in Bancilhon [1985] and Balbin and Ramamohanarao [1987], which we call *Basic Semi-Naive* (BSN) evaluation.

The BSN technique has the following phases. The first is a compilation phase that generates *semi-naive rules* from the given program. From each

rule in the program, a set of rules, which we call the *semi-naive versions* of the original rule, is generated. The second phase is performed at run-time; the evaluation keeps track of "new facts," that is, those that have not been used to make derivations. The evaluation proceeds in iterations; each iteration applies (semi-naive versions of) all program rules and makes all *new* derivations that can be made by using facts derived up to (and including) the previous iteration. Derivations using only facts generated two iterations (or more) earlier would have been made already, so only derivations that use at least one fact derived for the first time in the previous iteration are made. Some bookkeeping is performed at the end of each iteration to track which facts are new. This bookkeeping involves deletions from "differential" relations that keep track of which facts are new. Strictly speaking, deletions are not modeled in our definition of an evaluation. However, our notion of an evaluation is a high-level one, and semi-naive evaluations can be modeled using our definition by ignoring low-level details of the evaluation method.

States in a BSN evaluation have a one–one correspondence with iterations, where $\mathcal{F}_0$ is the set of EDB facts, and $\mathcal{F}_i = \mathcal{F}_{i-1} \cup$ the set of all facts derived in the ith iteration of BSN. For all i, $\mathcal{H}_i = \emptyset$, and facts are not discarded before the end of the evaluation.

For positive programs, BSN evaluation computes (in the limit) the meaning of a program as given by its least fixpoint semantics. BSN evaluation, as previously described, can be refined to work on a strongly connected component (SCC) of the program. The SCCs of a program are partially ordered, and the whole program is evaluated by evaluating the SCCs in a total order consistent with the partial order. This is referred to as SCC-by-SCC evaluation of the program.

Some of the techniques that we propose in this paper lead to evaluations that do not have the semi-naive property; however, they satisfy the following weaker notion.

Definition 2.2.2 (*Locally Semi-Naive Evaluation*). An evaluation is said to be a *locally semi-naive evaluation* if (a) no derivation step occurs more than once in any transition, and (b) for each derivation step D that appears in the ith transition, either: (1) D does not appear in any prior transition, or (2) there is a $j \leq i$ such that D uses at least one fact that is present in $\mathcal{F}_j - \mathcal{F}_{j-1}$, and derivation step D does not appear in any transition k, $j \leq k < i$.

Essentially a locally semi-naive evaluation is one where once a derivation step is carried out, it is not repeated unless one of the facts used in the derivation is discarded, and rederived subsequently.

To understand the intuition behind locally semi-naive evaluations, consider a variant of a BSN evaluation where a fact is deleted and later rederived. The fact would be considered a *newly derived* fact when it is rederived, and the evaluation may repeat some derivation that uses this fact. The evaluation would thus not be a semi-naive evaluation, but it would be a locally semi-naive evaluation.

PROPOSITION 2.2.3. *Every semi-naive evaluation is locally semi-naive. Every locally semi-naive evaluation is semi-naive if no facts are discarded during the evaluation.*

PROOF. The first part follows from the definitions, inasmuch as Parts (a) and (b1) in the definition of a locally semi-naive evaluation hold for every derivation step in a semi-naive evaluation.

For the second part, consider a locally semi-naive evaluation in which a derivation step D appears in transition i and also at some previous transition k. By the definition of locally semi-naive evaluations, there is a transition $j, k < j \leq i$ such that a fact used in D is derived in transition j, and is not available in the antecedent state of transition j. However, because D appeared in transition $k < j$, this fact must have been derived in some transition prior to k and therefore, it must have been discarded in some transition l, $k \leq l < j$. □

2.3 Base and Derived Predicates

In an evaluation, the facts for the EDB predicates are typically fixed for the duration of the evaluation. No new facts are derived for these predicates during the evaluation; hence EDB predicates are also referred to as "base" predicates (and predicates defined by rules are referred to as "derived" predicates). The knowledge that the set of facts in a "base" predicate is fixed is utilized during rule evaluation (e.g., this can simplify the form of semi-naive rewritten rules).

In some cases, predicates that are defined by rules can be treated as "base" predicates for the purposes of a number of optimization and evaluation methods. For instance, in an SCC-by-SCC BSN evaluation, predicates defined in lower SCCs can be treated as "base" in rules in higher SCCs. We present a generalized definition of "base" predicates, which is applicable for a variety of evaluation strategies.[6]

Definition 2.3.1 (*Base Set*). Consider an S-evaluation generated using an evaluation method $\mathcal{M}$, and a set of predicates $q_1, \ldots, q_n$. Let D_i be the set of all possible derivation steps that can be made using rule R in state $\langle \mathcal{F}_i, \mathcal{H}_i \rangle$. Similarly, let D'_i be the set of all possible derivation steps that can be made using R in state $\langle \mathcal{F}_i \cup \mathcal{Q}, \mathcal{H}_i \rangle$, where $\mathcal{Q}$ is the set of all facts for predicates $q_1, \ldots, q_n$ in the meaning of the program.

The set of predicates $q_1, \ldots, q_n$ is said to be a *base set* with respect to rule R if for each transition $\langle \mathcal{F}_i, \mathcal{H}_i \rangle \rightarrow \langle \mathcal{F}_j, \mathcal{H}_j \rangle$ where R is used to perform derivation steps, $D_i = D'_i$.

Note that a rule may have more than one base set; however, the union of base sets is not necessarily a base set. The following example illustrates this.

[6] This definition is strictly weaker than the usual definition. Hence, the use of the traditional definition does not affect the correctness of any of the results in this paper.

Example 2.3.2. Consider the following program.

$$
\begin{aligned}
&p(0).\\
&p(X+1) :- p(X), X < 100.\\
&q(0).\\
&q(X+1) :- q(X), X < 100.\\
&r(X) \quad :- p(X), q(X).
\end{aligned}
$$

In a BSN evaluation, where all program rules are applied in each iteration, $\{p\}$ is a base set with respect to the last rule. For example, consider the point in the semi-naive fixpoint evaluation when we have just derived $r(m)$, for some integer m. The set of available facts is $p(0), p(1), \ldots, p(m)$ and $q(0), q(1), \ldots, q(m)$. Even if all p facts in the least fixpoint of the program were available, that is, we had $p(0), p(1), \ldots, p(100)$, we could not derive $r(n)$, for $n > m$. Similarly, $\{q\}$ is a base set. However, the set $\{p, q\}$ is clearly not a base set.

Although the definition of base set is with respect to a particular evaluation, we only make choices that are correct with respect to any evaluation that can be generated by the evaluation method that we use. For example, in an SCC-by-SCC evaluation of the preceding program, both p and q would (trivially) be in the base set of the last rule of the program, because p and q would be "fully" evaluated before the first application of rule r.

Among all the base sets, one base set is chosen for each rule, and is referred to as the base set for the rule;[7] each predicate in it is said to be *base with respect to the rule*. The other predicates in the rule are said to be *derived with respect to the rule*.

A predicate p_2 is said to be *derived with respect to* another predicate p_1 if, either (1) there is a rule R such that p_1 is the head predicate of R and p_2 is derived with respect to R, or (2) there is a predicate p_3 such that p_3 is derived with respect to p_1 and p_2 is derived with respect to p_3.

A literal $p(\bar{t})$ in the body of rule R is said to be a *derived literal* (resp. *base literal*) if p is derived with respect to (resp. base with respect to) R. Note that the relation "derived with respect to" is not necessarily reflexive or symmetric.

3. ENSURING SOUNDNESS, COMPLETENESS, AND NONREDUNDANCY

In general, discarding a fact $p(\bar{a})$ in an evaluation could result in the nonderivation of other facts that would have been derived, had the fact $p(\bar{a})$ not been discarded. Thus, discarding facts could compromise completeness.[8] The following condition is the key for ensuring that facts are used in all possible derivations, and is used to ensure completeness of an evaluation.

[7] It is possible for predicates to be classified as being in a base set in a nonintuitive manner. However, such a choice does not affect the correctness of the results in this paper.

[8] In the presence of negation, discarding a fact could compromise soundness as well.

Condition U. Consider a point $e1$ in an evaluation E where a fact $p(\bar{a})$ is discarded. Fact $p(\bar{a})$ satisfies Condition U at $e1$ iff:

(1) Every derivation step using $p(\bar{a})$ has been made at or before $e1$ in E, or
(2) There is some later point $e2$ in E such that $p(\bar{a})$ is recomputed at $e2$ and any derivation step that can be made using $p(\bar{a})$ (given all the facts in the meaning of the program) but that has not been made before $e1$ is made after $e2$.[9]

The restriction of Condition U to a literal in the body of a rule is defined in a straightforward manner, by considering only uses of a fact in a particular literal in the body of that rule. It is straightforward to show that Condition U is satisfied by a fact $p(\bar{a})$ at a point $e1$ in an evaluation iff, for each body occurrence of p in every rule, the restriction of Condition U to that literal is satisfied by $p(\bar{a})$ in the evaluation.

If a fact is discarded and subsequently rederived, we may not detect the 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. If each fact that is discarded satisfies the following condition when it is discarded, then multiple derivations of facts will be detected, which can be used to ensure the semi-naive property:

Condition D. Consider a point $e1$ in an evaluation E where a fact $p(\bar{a})$ is discarded. Fact $p(\bar{a})$ satisfies Condition D at point $e1$ in E, iff:

(1) it is not derived again at or after the point $e1$ in E; or
(2) no derivation using $p(\bar{a})$ made at or before $e1$ is repeated after $e1$, even if $p(\bar{a})$ is rederived at some later point $e2$ in E.

The restriction of Condition D to a rule is defined in a straightforward manner, by considering only derivations of a fact by that rule. Again, it is straightforward to show that Condition D is satisfied by a fact $p(\bar{a})$ at a point $e1$ in an evaluation iff, for each rule defining predicate p, the restriction of Condition D to that rule is satisfied by $p(\bar{a})$ in the evaluation.

For an evaluation of a program to be complete, all nonredundant derivations that can be made using the program must in fact be made by the evaluation. In the case where the number of derivations of a fact do matter, as when the multiset semantics of Maher and Ramakrishnan [1989] is used, no derivation is redundant, although in other cases some derivations may be redundant. We say that an evaluation is *derivation-complete* if all derivations that can be made using the program are, in fact, made by the evaluation.

The following results summarize how ensuring Conditions U and D while discarding facts during an evaluation guarantees soundness, derivation-completeness, and the semi-naive property of the evaluation.

[9] If the program has negation, Condition U should also include the condition that any derivation that would have been prevented by the presence of $p(\bar{a})$ is not made in the evaluation.

Definition 3.1 (*Terminated Evaluation*). An evaluation of a program is said to be *terminated* if its last state[10] $\langle \mathcal{F}, \mathcal{H} \rangle$ is such that (a) $\mathcal{H} = \emptyset$, and (b) no fact $p \notin \mathcal{F}$ can be derived using the facts in $\mathcal{F}$ and any rule in the program.

PROPOSITION 3.1.1. *Consider a terminated* S*-evaluation E. Evaluation E is derivation-complete iff Condition* U *is satisfied by each fact whenever it is discarded.*

PROOF. (*If*): Let fact $p(\bar{a})$ be discarded at point $e1$ in the evaluation E. If $p(\bar{a})$ satisfies Condition U (1) at point $e1$ in E, then all derivations that use $p(\bar{a})$ have been made before $e1$ in E. If $p(\bar{a})$ satisfies Condition U (2) at point $e1$ in E, then those derivations that use $p(\bar{a})$ but have not been made before point $e1$, will be made after $p(\bar{a})$ is rederived in E. In either case, the evaluation E is derivation-complete.

(*Only if*): We prove the contrapositive. Assume that there is some fact $p(\bar{a})$ that does not satisfy Condition U when it is discarded at point $e1$ in the evaluation E. The fact $p(\bar{a})$ does not satisfy Condition U (1) at point $e1$, and hence not all derivations using this fact have been made at or before $e1$. Also, this fact does not satisfy Condition U (2) at point $e1$. Hence, exactly one of the following hold: (a) this fact is not recomputed after point $e1$ in the evaluation, or (b) this fact is recomputed in E after point $e1$, but there is at least one derivation using $p(\bar{a})$ which was not made before $e1$, and will not be made after $p(\bar{a})$ is recomputed. In either of these cases, evaluation E is not derivation-complete. □

PROPOSITION 3.1.2. *Consider a locally semi-naive* S*-evaluation E. Evaluation E has the semi-naive property iff Condition* D *is satisfied by each fact whenever it is discarded.*

PROOF. (*If*): If a fact $p(\bar{a})$ satisfies Condition D (1) when it is discarded at point $e1$ in evaluation E, no derivations that use $p(\bar{a})$ are made after $e1$ in E because E is locally semi-naive. Hence, to prove the semi-naive property of E, we need only consider rederivations of facts in E.

Our proof is by induction on the order in which facts are rederived after being discarded in E. Let $p_1(\overline{a_1})$ be the first fact rederived (say, at point $e2$) after being discarded (say, at an earlier point $e1$) in the evaluation E. Evaluation E is locally semi-naive, therefore no derivation step (including the derivation of $p_1(\overline{a_1})$ at $e2$) is repeated at or before point $e2$ in the evaluation E. Fact $p_1(\overline{a_1})$ must satisfy Condition D (2) at point $e1$ in E. Hence, derivations using $p_1(\overline{a_1})$ made before $e1$ in E are not repeated after $e2$. Also, there are no derivations using $p_1(\overline{a_1})$ between points $e1$ and $e2$ in E. Consequently, no derivation step that uses $p_1(\overline{a_1})$ is repeated in evaluation E.

Consider now the induction step. Let $p_n(\overline{a_n})$ be the nth fact rederived after being discarded in the evaluation E. The induction hypothesis guarantees

[10] The notion of last state is well defined only for finite evaluations. Our definitions can be extended to handle infinite evaluations as well, but the details are tedious.

that this derivation step is a new derivation step (because it has to use previously computed, possibly rederived, facts). Inasmuch as $p_n(\overline{a_n})$ must satisfy Condition D (2) when it is discarded, the argument used for the base case also shows that no derivation step that uses $p_n(\overline{a_n})$ is repeated in evaluation E. This completes the induction step and hence the proof of the claim that E has the semi-naive property.

(*Only if*): We prove the contrapositive. Assume that there is some fact $p(\bar{a})$ that does not satisfy Condition D when it is discarded at point $e1$ in the evaluation E. The fact $p(\bar{a})$ does not satisfy Condition D (1) at point $e1$, and hence it is derived again (say at point $e2$) in the evaluation E. Also, this fact does not satisfy Condition D (2) at point $e1$. Hence, some derivation made using $p(\bar{a})$ before $e1$ is repeated after $e2$ in E. Hence, evaluation E does not have the semi-naive property. □

The following result is a consequence of these two propositions.

THEOREM 3.1.3. *Consider a locally semi-naive, terminated* S*-evaluation* E. *Evaluation* E *is sound, derivation-complete, and has the semi-naive property iff Conditions* U *and* D *are satisfied by each fact whenever it is discarded.*

THEOREM 3.1.4. *Given a program* P, *and an arbitrary point* $e1$ *in an evaluation of program* P, *it is undecidable whether a given fact satisfies Conditions* U *and/or* D *at* $e1$.

PROOF. Consider an arbitrary logic program L that defines p. Add the fact p_1 and the rule $p_2 :- p, p_1$ to L to get program L1. (Neither p_1 nor p_2 should occur in L.) The fact p_1 can be used to compute p_2 iff $?p$ is satisfiable in L. Because satisfiability is not decidable for logic programs [Shmueli 1987], it is undecidable if p_1 will be used again after any point e_1 in the evaluation. Because there is no other derivation of p_1, Condition U is undecidable.

To show undecidability of Condition D, add the fact p and the rule R: $p_1 :- p$ to the logic program L to get L2. Consider a point e_1 in a semi-naive evaluation of L2 after p_1 has been derived using the given fact p and rule R. Because it is undecidable whether $?p$ is satisfiable in L, if p is discarded at e_1 it is undecidable whether the derivation of p_1 (using R) is repeated after e_1. □

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 Conditions D and U for program facts. Even the stronger conditions that test only 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.

4. OUR APPROACH TO SPACE OPTIMIZATION

4.1 Discarding Facts Based on Conditions U and D

The general framework of our approach for discarding facts during an evaluation of a program is as follows.

Consider an evaluation method. The evaluation method performs certain tests before discarding facts during an evaluation. Facts are discarded only if they satisfy these tests. Some of these tests guarantee that if a fact satisfies the test when is discarded, then, in the evaluation generated by the evaluation method, the fact satisfies Condition U when it is discarded. We call such tests *techniques for ensuring Condition* U. Similarly, we have *techniques for ensuring Condition* D.

Each technique we describe for ensuring Conditions D and/or U is typically applicable only to certain classes of programs. Hence, at compile time we analyze the program, and decide on the applicability of each technique for ensuring Conditions D and U. We then generate a specific evaluation method for a program by choosing which techniques to use. These techniques perform tests at run-time to decide when a fact satisfies Conditions D and/or U. Facts are discarded at run-time as soon as the tests determine that they satisfy Condition U and (if nonredundancy is desired) Condition D. The run-time tests we describe are quite efficient—see Section 9 for more details.

The discarding of one fact at a point in the evaluation could affect whether another fact is rederived at a later point in the evaluation. Hence, for simplicity, we have assumed that facts are discarded one at a time in the evaluation. The run-time tests for whether a given fact satisfies Conditions D and U are performed under the assumption that no other fact is discarded at the same point in the evaluation.

4.2 Summary of Our Techniques

In the rest of the paper, we describe several techniques for ensuring Conditions D and U, as well as several synchronization techniques.

Ensuring Condition D (*Nonredundancy*). Techniques for ensuring Condition D can be chosen 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 [Maher and Ramakrishnan 1989], we know that once a fact is derived it will not be derived again. We look at this technique (and some extensions) for ensuring Condition D in Section 5.1.

(2) Using monotonicity constraints.
Monotonicity constraints ensure some monotone ordering on the derivation of facts. We look at this idea in Section 6.3.

Ensuring Condition U (*Correctness*). Techniques for ensuring Condition U can be chosen 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, that is, there is only one literal in the body of the rule whose predicate is derived with respect to the rule. Once a fact for

the derived predicate is used (along with all the facts for the "base" predicates in that rule), 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 5.2.

(2) Using monotonicity constraints.
In Section 6.4 we consider using monotonicity constraints to ensure Condition U.

If none of these approaches for ensuring Conditions D or U succeeds for p facts, we always have the option of not discarding any p facts. We can still optimize the rest of the program, unlike the method described in Naughton and Ramakrishnan [1994].

Synchronization. If (all) derivations of facts are "close" to all their uses, facts can be discarded soon after being derived (and used). In Section 7 we consider techniques that can be used to order an evaluation to ensure that derivations of facts are close to their uses, and we call them *synchronization* techniques. These include:

—*Delaying first use of facts*: The idea is to partition the set of derived facts into a set of "active" facts used in derivations and a set of "hidden" facts whose use is delayed (until they become "active"). 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 point in the evaluation is reduced (Section 7.1).

—*Nested-Unit synchronization*: This technique identifies "subgoals" that are to be evaluated by a "subprogram" on each call. The idea is to generate (answer) facts using the subprogram as and when they are needed by the "main" program (Section 7.2).

—*Interleaved-Unit synchronization*: The acyclic graph of SCCs of a program suggests a natural producer/consumer relationship. By interleaving the evaluation of producers and consumers, it is sometimes possible to ensure that facts are generated in a producer as and when they are needed by the consumers. This is generalized to work with "units" instead of SCCs (Section 7.3).

Combining Techniques. The various techniques for synchronization and for ensuring Conditions D and U are applicable to parts of a program (such as rules, literals, etc.). These need to be combined to get a space optimization method for the full program. This issue is discussed in some detail in Section 8.2.

5. BOUNDS ON DERIVATIONS AND USES OF FACTS

In the following, we use the notion of functional dependencies on variables and literals in rule bodies.

Definition 5.1 (*Functional Dependencies on Rules*). We say that a set of variables $\overline{X}$ in a rule R *functionally determines* a set of variables $\overline{Y}$ in R if

the following condition is met: given any rule instances $R[\theta]$ and $R[\gamma]$ such that the bodies of $R[\theta]$ and $R[\gamma]$ are satisfied in the meaning of the program, if $\bar{X}[\theta] = \bar{X}[\gamma]$, then $\bar{Y}[\theta] = \bar{Y}[\gamma]$.

A set of literals $\bar{p}$ in R functionally determines a set of literals $\bar{q}$ in R if $vars(\bar{p})$ functionally determines $vars(\bar{q})$.

The preceding notation is abused to allow single variables or literals in place of sets of variables or literals.

5.1 Duplicate Freedom and Condition D

The simplest technique to ensure Condition D for a fact $p(\bar{a})$ at a point in a locally semi-naive evaluation of a program is based on the following condition on the predicate p:

Condition DF1. A predicate satisfies condition DF1 if: (1) No fact for p is derived by more than one rule, and (2) there is at most one derivation for each p fact by any rule.

The essential idea is to make sure that no fact for p is derived more than once in the evaluation. The techniques of Maher and Ramakrishnan [1989] can be used to test the condition—part (1) can be tested by determining that no two rule heads unify[11] and part (2) by checking that the head of a rule functionally determines the body of the rule.

PROPOSITION 5.1.1. *If a predicate p in a locally semi-naive* S*-evaluation satisfies Condition* DF1, *and the evaluation is semi-naive with respect to p, Condition* D *is satisfied by each p fact at any point at which it is discarded.*

PROOF. Conditions DF1 (1) and DF1 (2) together ensure that there is at most one derivation of any p fact. Because the evaluation is semi-naive with respect to p, such a derivation step is not repeated. Hence, predicate p straightforwardly satisfies Condition D (1). □

We can weaken Condition DF1 in several ways. If part (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.

DF1 can also be weakened by modifying the requirement that "there is at most one derivation for each fact by any rule" to the requirement that "if there is more than one derivation for any fact by a rule, then the facts for the derived literals in the corresponding rule instances are the same." Thus multiple derivations would be allowed within a rule application. To test this weaker requirement, we can check whether the head of a rule functionally determines all derived body predicate occurrences in the rule; the head need not functionally determine the base predicate occurrences. To summarize:

Condition DF2. Part (1) as in DF1, and (2) if there is more than one derivation for any fact by a rule, then the facts for the derived predicate occurrences in the corresponding rule instances are the same.

[11] This can be generalized using the techniques of Debray and Warren [1989], for example.

A proposition similar to Proposition 5.1.1 also holds in the case of DF2. Again, if part (1) does not hold, we can ensure Condition D using run-time checks.

5.2 Bounds on Uses and Condition U

If we can determine a bound on the number of uses of p facts in a body predicate occurrence p' of p, once a p fact has been used in that many derivations in p', we know that it can no longer be used in this occurrence. The following condition seeks to capture this intuition.

Condition Bounds–U. Consider a program P, and a rule R which has a body predicate occurrence $p(\bar{t})$. Let R be denoted as:

$$R\colon p2(\overline{t2}) :- p(\bar{t}), b(\overline{t0}), p1(\overline{t1}),$$

where $b(\overline{t0})$ denotes the set of all the predicate occurrences in R (other than $p(\bar{t})$) that are base with respect to the rule, and $p1(\overline{t1})$ denotes the set of all predicate occurrences (other than $p(\bar{t})$) that are derived with respect to the rule.

Then the predicate occurrence $p(\bar{t})$ in rule R satisfies Condition Bounds–U if it satisfies either of:

BU1: $p(\bar{t})$ functionally determines $p1(\overline{t1})$ in R, or

BU2: $p(\bar{t})$ functionally determines the head $p2(\overline{t2})$ of R.

PROPOSITION 5.2.1. *Consider a locally semi-naive* S*-evaluation of a program P, and a rule in P:*

$$R\colon p2(\overline{t2}) :- p(\bar{t}), b(\overline{t0}), p1(\overline{t1}).$$

Suppose body predicate occurrence $p(\bar{t})$ in R satisfies Condition BU1 *and the evaluation is semi-naive with respect to $p2$. Then no derivation step in any transition after a point $e1$ will use fact $p(\bar{a})$ in the body literal $p(\bar{t})$ if*: (1) *there is no instance $R[\theta]$ whose body is satisfied in the meaning of the program and $p(\bar{t})[\theta] = p(\bar{a})$, or* (2) *an application of R that uses a derivation step where $p(\bar{t})$ is instantiated to $p(\bar{a})$ has been made at or before $e1$.*

PROOF. If no instance $R[\theta]$ whose body is satisfied in the meaning of the program is such that $p(\bar{t})[\theta] = p(\bar{a})$, then $p(\bar{a})$ cannot be used in any derivation step; this proves Part (1) of the result. If the fact has been used in a derivation step, all derivation steps that it can be used in must have been carried out in the same rule application by the definition of base with respect to a rule. Because the evaluation is semi-naive with respect to $p2$, none of these derivation steps can be repeated in the evaluation; this proves Part (2) of the result. □

Consider the (important) special case of linear recursive rules. For such rules, Condition Bounds–U is always satisfied—$p(\bar{t})$ is the only derived predicate occurrence in the rule R and BU1 is satisfied trivially. If we use BSN evaluation, any p fact is either used in a derivation step using R in the

iteration after it is derived, or there is no instantiation of the rule R with this p fact, such that the body is satisfied in the meaning of the program. Consequently, by the preceding proposition, every p fact satisfies Condition U if it is discarded at any point after the end of the iteration subsequent to the iteration in which it is derived.

Now suppose some derivations using R and $p(\bar{a})$ need to be repeated. If we discard the fact $p(\bar{a})$ after a derivation step, we would prevent repetitions of that derivation and hence not satisfy Condition U.

If the occurrence $p(\bar{t})$ in R satisfies Condition BU2, and a fact $p(\bar{a})$ has been used in this occurrence in a derivation step, then no new facts can be generated by any subsequent derivations that instantiate $p(\bar{t})$ to $p(\bar{a})$. Condition U is not satisfied if the fact is discarded, as it is possible that there are other derivations of the *same* head fact using $p(\bar{a})$ in this predicate occurrence. However, if we are not interested in the number of derivations, we may effectively consider Condition U to be "satisfied," without compromising soundness or completeness (although we do sacrifice derivation-completeness).

The following result is a direct consequence of the preceding proposition.

COROLLARY 5.2.2. *Suppose a literal $p(\bar{t})$ in the body of rule R satisfies Condition Bounds_U, and in a point in a locally semi-naive* S*-evaluation a fact $p(\bar{a})$ satisfies the tests described in Proposition* 5.2.1. *Then $p(\bar{a})$ satisfies the restriction of Condition* U *to the literal $p(\bar{t})$ if it is discarded at that point.*

The following example illustrates the use of bounds on derivations and uses of facts in space optimization.

Example 5.2.3. Consider the program for computing the ancestors of a given person and the *father* relation, from Example 1.2.1.

$$anc(X, Y) :- father(X, Y).$$
$$anc(X, Y) :- father(X, Z), anc(Z, Y).$$
Query: ?-$anc(n, X)$.

Suppose the *father* relation is an acyclic relation, with the functional dependency *father*: \$1 → \$2, that is, each person has at most one father.

We can deduce the following about the program:

—The body occurrence of $anc(Z, Y)$ in the linear recursive rule satisfies Conditions Bounds_U. Hence, in an evaluation of the program, each *anc* fact can be used in at most one rule application (although, possibly, in several derivation steps), in the iteration subsequent to the iteration in which it is derived.

—The definition of *anc* satisfies Condition DF1, that is, no *anc* fact is derived more than once. The functional dependency between the arguments of the *father* relation shows that each *anc* fact can be derived at most once by each rule, and the acyclicity of the *father* relation along with the functional dependency shows that each fact can be deduced by at most one rule. (Techniques such as those presented in Maher and Ramakrish-

nan [1989] may be used to deduce this; this is outside the scope of our paper.)

Hence, we can discard *anc* facts that are not answers to the query at the end of the iteration after they are computed.

Consider a *father* relation with the following facts: $father(n, n-1)$, $father(n-1, n-2), \ldots, father(2,1)$, $father\ (1,0)$. A bottom-up evaluation of the Magic Sets transformed program would have asymptotic space and time complexity of $O(n^2)$. This is because the evaluation computes the following *anc* facts: $anc(n, n-1), \ldots, anc(n,1), anc(n,0), anc(n-1, n-2), \ldots, anc(n-1,1), anc(n-1,0), \ldots, anc(1,0)$. In an SCC-by-SCC evaluation of the Magic Sets transformed program, we can discard *anc* facts one iteration after being derived. This results in an $O(n)$ space complexity. (The time complexity remains unchanged.)

Note that the use of functional dependencies is conservative; for example, if literal p functionally determines literal q in the body of a rule R, we know that there is at most one q fact that can be used in a derivation step with a given p fact. In fact, there may be no such q fact (as the following example illustrates). Conditions BU1 and BU2 for the literal p can be refined by using a notion of dependencies that requires the existence of exactly one such q fact for each p fact, but we do not pursue the extension further.

Example 5.2.4. Consider the following program, P_{Ack}, that computes the Ackermann function:

$R1$: $ack(0, Q, 2 * Q)$.
$R2$: $ack(P, 0, 0) :- P > 0$.
$R3$: $ack(P, 1, 2) :- P > 0$.
$R4$: $ack(P, Q, N) :- P > 0, Q > 1, ack(P, Q-1, N1), ack(P-1, N1, N)$.

Here, the FDs *ack*: $\{\$1, \$2\} \rightarrow \$3$ and $\{\$1, \$3\} \rightarrow \$2$ hold.[12] As a consequence, each body occurrence of *ack* functionally determines the other occurrence of *ack* in rule $R4$. Also, each occurrence of *ack* in the body of rule $R4$ functionally determines the head of that rule. Both Conditions BU1 and BU2 are therefore applicable; because *ack* can also be shown to be duplicate-free, we can discard each *ack* fact after two uses. However, there are several *ack* facts that can be used only once, and based on these conditions they are never discarded.

6. 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 Naughton and Ramakrishnan [1994].

[12] The FD *ack*: $\{\$1, \$2\} \rightarrow \$3$ holds because the third argument denotes the result of the Ackermann function on the first two arguments. The FD *ack*: $\{\$1, \$3\} \rightarrow \$2$ holds because (it can be easily shown that) for a given value of the first argument, the value of third argument is strictly monotonic on the value of the second argument.

6.1 ϕ Functions

We make extensive use of the class of ϕ functions defined in Naughton and Ramakrishnan [1994]; we present it now for completeness. The ϕ functions are similar to the "size" functions that have been used for detecting termination of logic programs.

We let Φ denote a class of functions that map ground atoms to integers ($\mathcal{I}$). Individual functions in the class Φ are typically denoted as ϕ, and we often refer to Φ as the class of ϕ functions.

Consider a predicate $p(X_1, X_2, \ldots, X_n)$, where $\mathcal{D}_i$ denotes the domain of X_i.[13] A function ϕ that is applied to atoms of predicate p can then be viewed as a function:

$$\mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n \rightarrow \mathcal{I}.$$

For simplicity, we consider only functions that can be expressed as arithmetic expressions on the variables $X_1, X_2, \ldots, X_n$, possibly with predefined functions (such as term-size) applied to the variables. An example of a function in Φ is one that maps $fac(I, N)$ to I.

Intuitively, ϕ functions are used to formalize monotonicity that is present in the use or generation of facts. Monotonicity can be used to discard facts once they are no longer needed. In order to characterize the monotonicity in the generation or use of facts for a given predicate, we choose a ϕ function for that predicate. There are of course many candidate functions from which to choose.

An important question is, how do we choose such a function automatically. In general, the task is quite difficult. In this paper, we deal with the issue as follows. We enumerate all choices from the subclass of ϕ functions that sum up the "sizes" of a subset of the arguments of the predicate. (We define the "size" of an argument later.) From this set of functions, we choose a function that lets us infer "monotonicity of rules." (We discuss the testing for monotonicity later.) If there are multiple functions that enable us to infer monotonicity, we make an arbitrary choice between them. The specific choice made can certainly affect the efficiency of the space optimization method. How precisely to make such a choice is outside the scope of this paper.

We now present the definition we use for the *size* of a term; other definitions are possible, and our choice is merely a convenient one. To define the *size* of a term, we divide terms into three types: those that contain integers, those that contain structured terms, and others. The "size" of an argument is defined as follows:

(1) The size of an integer is itself.

(2) The size of a structured term $f(t_1, t_2, \ldots, t_p)$ is defined by

$$size\big(f(t_1, t_2, \ldots, t_p)\big) = 1 + \max\big(size(t_1), size(t_2), \ldots, size(t_p)\big).$$

(3) The size of a term that is neither an integer nor a structured term is 1.

[13] In this section we assume that we know the domain from which facts take values. If the domain is not known, we simply assume that it is the universe.

For example, suppose that the function ϕ chosen for a predicate $p(X, Y, Z)$ is $size(X) + size(Y)$. If we know that the first two arguments are integers, the function ϕ merely becomes $X + Y$. If we do not know the types of arguments a priori, the computation may proceed as shown in the following:

$$\begin{aligned}\phi(p(f(g(c), a), 4, b)) &= size(f(g(c), a)) + 4 \\ &= 1 + \max(size(g(c)), size(a)) + 4 \\ &= 1 + \max(1 + size(c), 1) + 4 \\ &= 3 + 4 \\ &= 7.\end{aligned}$$

Applying ϕ to a structured fact (such as a fact with lists) could take time proportional to the size of the arguments of the fact. However, it may be possible to compute the ϕ value for a fact incrementally, by carrying along additional information during derivations. For instance, consider the following program that appends two lists:

$$append(nil, L, L).$$
$$append([H \mid T], L, [H \mid L1]) :- append(T, L, L1).$$

If we use a function ϕ that sums the sizes of the first two arguments of *append*, it is possible to compute the ϕ value for a newly derived *append* fact incrementally by using the ϕ value of the *append* fact in the body of the recursive rule. We do not discuss this issue further in the paper.

The cost of complete enumeration of the ϕ functions in the aforementioned subclass, for a single predicate, is exponential in the number of arguments of the predicate; with multiple predicates we have to multiply together the number of choices for each predicate in order to enumerate all possible choices. However, the number of arguments of a predicate is typically small and because we analyze program units (Section 8.2) with only a few predicates, we expect that the cost will be reasonably small in practice. A heuristic that we have found useful is the following. We analyze the "mode" (or, equivalently in the context of evaluation using Magic Sets, the adornment) in which a predicate is used, and choose as a ϕ function the sum of the sizes of all the input arguments (equivalently, "bound" arguments).

6.2 Local Saturation

Consider a Basic Semi-Naive evaluation, and any point in that evaluation. We know that some facts are "old" and other facts are "new," and each new derivation step must use at least one new fact. We may be able to infer that any fact that is derived after this point in the evaluation will have a ϕ value greater than any "new" fact that is available, and thereby deduce which facts will not be derived again, and which facts will not be used again.

This intuition behind our use of monotonicity information is not limited to Basic Semi-Naive evaluation. We would like to use monotonicity information, independent of the specific evaluation method used. Hence, we use the following definition of "locally saturated" facts, which corresponds to the "old"

facts in BSN; the idea is that each new derivation step must use at least one fact that is not in the set of "locally saturated" facts.

Definition 6.2.1 (*Locally Saturated*). A set of facts S for derived body predicate occurrences of a rule R defining p is said to be *locally saturated with respect to* R at a point $e1$ in the evaluation if every derivation step that can be made using: (1) the rule R, (2) all facts for the base predicate occurrences in R, and (3) the given set of facts S for the derived predicate occurrences in R, has been made at or before $e1$.

A set of facts is said to be *locally saturated with respect to a set of rules* at a point in the evaluation if it is locally saturated with respect to each of the rules at that point in the evaluation.

Note that there can be more than one set of locally saturated facts at a state in an evaluation. Because all derivations that could be made using a set of locally saturated facts have been made at a point in the evaluation, any new derivation after that point in the evaluation requires at least one fact (for a derived predicate occurrence) that is not in the set of locally saturated facts.

In the case of a Basic 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 in or before the $(n - 1)$th iteration is a set of locally saturated facts for p. If a different evaluation technique is used, the sets of locally saturated facts may change, but each new derivation would still have to use at least one fact that is not in the set of locally saturated facts. Thus, we achieve a certain degree of independence from specific evaluation techniques in the following results.

6.3 Monotonicity and Condition D

Definition 6.3.1 (*Monotonicity*). A rule

$$R\colon p(\bar{t}) :- \ldots, p_i(\bar{t_i}), \ldots$$

is said to be *monotonically increasing with respect to predicate occurrence* $p_i(\bar{t_i})$ in its body if, for every instance $R[\theta]$ of the rule where the body is satisfied in the meaning of the program, $\phi(p(\bar{t})[\theta]) \geq \phi(p_i(\bar{t_i})[\theta])$.[14] A rule is said to be *monotonically increasing* if it is monotonically increasing with respect to all body occurrences of predicates that are derived with respect to the rule.

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

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

[14] It would probably be better to use the term "inflationary" rather than "monotonicity." However, we use "monotonicity" for consistency with earlier work in this area [Naughton and Ramakrishnan 1994].

The rule is guaranteed to be monotonically increasing if for each derived literal $p_i(\bar{t}_i)$, the arithmetic expression $\phi(p(\bar{t})) - \phi(p_i(\bar{t}_i))$ is always nonnegative. This can be tested using symbolic manipulation on each expression $\phi(p(\bar{t})) - \phi(p_i(\bar{t}_i))$, along with the arithmetic literals in the rule body.

Condition Monotonicity–D. Consider a program P. Let p be a predicate defined in P, and let the set S include p and the set of all predicates in P that are derived with respect to p. Let $\mathscr{R}$ be the set of all the rules of P defining the predicates in S. The predicate p satisfies Condition Monotonicity–D iff each rule in $\mathscr{R}$ is monotonically increasing.

Definition 6.3.2 (*Min-Head-Gap Bounding Function*). For a predicate p satisfying Condition Monotonicity–D, a function γ mapping ground atoms to integers is said to be a *min-head-gap bounding function for* p if for each instance R' of any rule R defining p, if $p(\bar{a})$ is the head fact and $q(\bar{b})$ is the fact for any derived literal in the body of R', $(\phi(p(\bar{a})) - \phi(q(\bar{b}))) \geq \gamma(q(\bar{b}))$.

Note that the constant function $\gamma = 0$ is always a min-head-gap bounding function for such predicates—however, one might be able to determine a "better" function for the purpose 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 $p_i(\bar{t}_i)$ in the body of R, each expression $\phi(p(\bar{t})) - \phi(p_i(\bar{t}_i))$ not only is nonnegative but also (after simplification) has as arguments only variables from $\bar{t}_i$. Then we can derive a min-head-gap bounding function for p by symbolic arithmetic manipulations on these functions. For instance, if we have the rule

$$fac(X, X * N) :- X > 0, Y = X - 1, fac(Y, N),$$

and a ϕ function that maps $fac(X, Y)$ to X, simplification of $\phi(X) - \phi(Y)$ using $Y = X - 1$ gives us the constant function 1 as a min-head-gap bounding function for *fac*.

THEOREM 6.3.3. *Consider a locally semi-naive* S*-evaluation where predicate p satisfies Condition Monotonicity*–D, *the evaluation is semi-naive with respect to p, and* γ *is a min-head-gap bounding function for p. Let S and* $\mathscr{R}$ *be as in Condition Monotonicity*–D. *In this evaluation, let F be the set of all the facts that have been computed 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* $\mathscr{R}$. *Let*

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

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

PROOF. The set of facts F' is locally saturated with respect to the set of rules $\mathscr{R}$. Hence, for any predicate $q \in S$ (i.e., p, or a predicate derived with respect to p) any derivation of a q fact using some rule $R \in \mathscr{R}$ must use at least one fact that is not in F'. Because the rules in $\mathscr{R}$ are monotonic, for any new fact $q(\bar{b})$, $\phi(q(\bar{b})) \geq \min\{\phi(f) \mid f \in F - F'\}$. Because γ is

a min-head-gap bounding function for p, no p fact with a ϕ value less than $\min\{\phi(f) + \gamma(f) \mid f \in F - F'\}$ will be derived. □

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 derived two or more iterations prior to the current iteration constitutes F' (as mentioned earlier) and the set of facts derived either in the previous or in 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 specific evaluation.

Example 6.3.4. Consider the following program that computes a list of factorials of even integers, and an iterative evaluation of all the rules in the program.

$R1$: $fac\_list(0, [1])$.
$R2$: $fac\_list(N, [V \mid L]) :- N > 0, N < n, fac\_list(N - 1, L), fac(2 * N, V)$.
$R3$: $fac(0, 1)$.
$R4$: $fac(N, N * V) \quad :- N > 0, N < 2 * n, fac(N - 1, V)$.

Let the ϕ function map $fac\_list(N, \_)$ to N, and $fac(N, \_)$ also to N. We deduce that rules $R3$ and $R4$ are monotonically increasing. In rule $R2$, *fac* can be treated as "base." 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 6.3.3 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*.

6.4 Monotonicity and Condition U

In this section we discuss how to use monotonicity of rules to ensure Condition U. We make use of the definitions and results in Section 6.3. Let ϕ be a function as before.

Definition 6.4.1 (*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:

$$body\_gap(R', q') = \max\{body\_gap(R', p', q') \mid p' \text{ is a derived predicate occurrence in } R\}.$$

If R has no derived predicate occurrence in its body, $body\_gap(R', q') \doteq \infty$.

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 6.4.2.

Condition Monotonicity_U. Consider a program P. Let R be a rule with a body predicate occurrence q' having predicate 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 $q(\bar{a})$ used in the occurrence q'):

$$body\_gap(R', q') \leq \gamma(q(\bar{a})).$$

Intuitively the theorem states that if two facts are used in a rule to perform a derivation step, 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 6.3, we can derive a function γ as in Condition Monotonicity_U.

THEOREM 6.4.2. *Consider a point in a locally semi-naive* S*-evaluation of a program P. Let R be a rule in P, q' be a body predicate occurrence in R, and $p'_1, \ldots, p'_n$ be the derived predicate occurrences in the body of the rule 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.*[15] *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 at the given point in the evaluation.*

Then, every derivation step that instantiates predicate occurrence q' to $q(\overline{a_1})$ must have been made before the given point in the evaluation if $\phi(q(\overline{a_1})) + \gamma(q(\overline{a_1})) < m$.

If the evaluation is semi-naive with respect to the head predicate of R, the fact $q(\overline{a_1})$ will not be used in the predicate occurrence q' after the given point in the evaluation.

PROOF. Because the set of 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, and no p_i fact with ϕ value less than m will be derived again, any new derivation must use at least one derived fact with ϕ value of m or more. But by Condition Monotonicity_U if a fact $q(\overline{a_1})$ is used to make a derivation, all derived facts used with it are such that their ϕ values are less than or equal to $\phi(q(\overline{a_1})) + \gamma(q(\overline{a_1}))$. Hence, a fact $q(\overline{a_1})$ will not be used in the predicate occurrence q' beyond this point in the evaluation if $\phi(q(\overline{a_1})) + \gamma(q(\overline{a_1})) < m$. □

[15] Theorem 6.3.3 may be used to ensure this

Note that the theorem makes no mention of whether q is derived with respect to the head of the rule. 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). Theorem 6.4.2 generalizes the conditions of Sliding Window Tabulation [Naughton and Ramakrishnan 1994], as only such constant functions could be used for γ in Sliding Window Tabulation. Example 6.4.3 shows the importance of allowing general functions.

Example 6.4.3 We use the program from Example 6.3.3 again. Consider Rule $R4$:

$$R4\text{: } fac(N, N*V) :- N > 0, N < 2*n, fac(N-1, V).$$

There is only one derived literal in the body of this rule, hence a *fac* fact can be used at most once in this rule (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 in the following:

$$R2\text{: } fac\_list(N, [V \mid L]) :- N > 0, N < n, fac\_list(N-1, L), fac(2*N, V).$$

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 $fac(2*N, \_)$ to $N - 1 - 2*N$, and hence $fac(M, \_)$ to $M/2 - M - 1$. 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 that have not been used to make derivations, then *fac* facts $fac(M, \_)$ such that $M + (M/2 - M - 1) < n$ will no longer be used. But from Example 6.3.3 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 Basic Semi-Naive evaluation of the program where all program rules are applied in each iteration, one iteration after $fac\_list(n, \_)$ has been produced we know that any $fac(m, \_)$ fact with $m/2 - 1 < n$ can no longer be used in the occurrence of *fac* in rule $R2$.

7. SYNCHRONIZATION

A synchronization technique orders derivations in the evaluation of a program so that derivations of facts are "close" to their uses; this helps reduce the "lifetimes" of facts. Intuitively, if each fact computed in an evaluation is stored for only a short while during the evaluation, the total space required for the overall evaluation can be reduced. We begin with an example where synchronizing helps in improving the space utilization of a program evaluation. In the rest of this section, we present three techniques for achieving synchronization.

Example 7.1. Consider the following program, where n is some constant, and an iterative BSN evaluation of all the rules in the program.

$R1$: $fac(0, 1)$.
$R2$: $fac(N, N * X1)$ $:-N > 0, fac(N - 1, X1)$.
$R3$: $fac1(n, 1)$.
$R4$: $fac1(N, N * X1)$ $:-N > n, fac1(N - 1, X1)$.
$R5$: $fac2(n, 1)$.
$R6$: $fac2(N + 1, Y * Y1 * Y2)$ $:-N > n, fac2(N, Y), fac(N, Y1), fac(N, Y2)$.

Query: ?-$fac2(m, X)$.

Let us consider the case when $m \geq n$; for $m < n$, the answer set to the query is empty. If each fact in this evaluation is used as soon as it is derived (or in the following iteration as when Basic Semi-Naive evaluation is used), we would have to store $n + 6$ facts at any point in the evaluation (from the $n + 1$th iteration onward, although less in previous iterations) based on satisfaction of Conditions U and D. However, if all uses of a $fac1(N, \_)$ fact are delayed until $fac(N, \_)$ has been derived, we need store only six facts at a point in the evaluation. Because n can be arbitrarily large, synchronizing the evaluation helps considerably in improving the space utilization of the program evaluation.

In order to handle the complexity of choosing a synchronization technique for a given (possibly large) program, we partition the rules of the program into subprograms which we call *units*, then decide on the synchronization techniques to be used between units. We describe how to partition the rules of a program into units in Section 8.2. The applicability of synchronization techniques depends on semantic properties of this partitioning, and we describe these properties when presenting the various synchronization techniques.

An example of the partitioning of a program into units is the partitioning defined by the SCCs of the program; each SCC contains a maximal set of mutually recursive predicates, along with the set of rules defining the predicates.

7.1 Delaying First Use of Facts

An integral part of the Sliding Window Tabulation technique for space optimization [Naughton and Ramakrishnan 1994] is the idea of keeping all uses of a derived fact "close" together in the evaluation—this is done by delaying the first use of a (derived) fact. Each fact is assigned an integer index by a ϕ function. At each point in the evaluation, there is an active "window" of facts; a fact whose index is not in this window is not available for immediate use in rule applications—it is *hidden* and can be used only when its index falls in the current window. In this section, we generalize this idea of Naughton and Ramakrishnan [1994] and see how it helps in synchronization of evaluation.

Condition Hiding_Facts. A unit satisfies Condition Hiding_Facts if:

(1) All the rules of the unit are monotonically increasing.
(2) There exists a function γ' mapping facts to integers such that for each derived body predicate occurrence p' of an instance R' of a rule in the unit, where fact $p(\bar{b})$ is used in p', $body\_gap(R', p') \leq \gamma'(p(\bar{b}))$.
(3) There is a finite bound min_ϕ such that $\phi(p(\bar{b})) \geq min_\phi$ for all facts $p(\bar{b})$ for each predicate p defined in the unit.

PROPOSITION 7.1.1. *Consider an* S*-evaluation of a unit that satisfies Condition Hiding_Facts. Let m_D be any integer. Let F be the set of $p(\bar{b})$ facts for which $\phi(p(\bar{b})) + \gamma'(p(\bar{b})) = m_D$. Facts $q(\bar{c})$ with $\phi(q(\bar{c})) \leq m_D$ must be made available to rule applications, for facts in F to be completely used. $q(\bar{c})$ facts with $\phi(q(\bar{c})) > m_D$ cannot be used along with any fact from F in any rule application.*

PROOF. Because for rule instance R' (with $p(\bar{b})$ used in predicate occurrence p') $body\_gap(R', p') \leq \gamma'(p(\bar{b}))$, any derived fact used in R' must have a ϕ value less than or equal to $\phi(p(\bar{b})) + \gamma'(p(\bar{b}))$. Hence for any fact p in the set F, if a derived fact q is used in an instance of a rule in the unit along with p, then $\phi(q) \leq m_D$. Facts with greater ϕ values cannot be used in a rule application with any fact from F. □

7.1.1 *Evaluation With Hiding Facts.* Proposition 7.1 provides a basis for the *hiding of facts* to reduce space utilization. Consider a unit S that satisfies Condition Hiding_Facts. The value min_ϕ may be determined in one of several ways: it may be determined by program analysis (as in Example 7.1); or, if S is an SCC in a Magic rewritten program and the Magic predicates corresponding to the predicates in S are in a lower SCC, it may be determined based on an evaluation of the SCC containing the Magic predicates.[16]

At a point in the evaluation of S, let m_D be the greatest integer such that the set of all program facts with ϕ values $< m_D$ is locally saturated with respect to all the rules in the unit. Initially, m_D is set to min_ϕ. Because the unit S has monotone rules, the value of m_D can be determined at later points in the evaluation as discussed in Section 6.3. We modify the evaluation of S by always hiding derived facts with indices greater than m_D. (The hidden facts are part of the hidden component of a program evaluation state.) The value m_D could increase each time facts are derived; it can be updated, for instance, at the end of each iteration in a Basic Semi-Naive evaluation of the unit.

Extensions of BSN evaluation to handle hiding of facts, and rule ordering are presented in Ramakrishnan et al. [1994]. The details do not concern us in this paper.

To see how delaying the first use of facts can improve space utilization, consider $q(\bar{c})$ facts with $\phi(q(\bar{c})) > m_D$. Any $p(\bar{d})$ fact that can be used in a

[16] This can be generalized to work with units, instead of SCCs, in a straightforward fashion.

rule application with such a $q(\bar{c})$ fact would have $\phi(p(\bar{d})) + \gamma'(p(\bar{d})) > m_D$. Because facts with a ϕ index of m_D can still be derived, such a $p(\bar{d})$ fact cannot be discarded at this point in the evaluation based on Theorem 6.4.2 to ensure Condition U. If these $q(\bar{c})$ facts are used along with $p(\bar{d})$ facts in a rule application, new facts can be derived but none of the ($p(\bar{d})$ or $q(\bar{c})$) facts used to derive these new facts can be discarded. By hiding $q(\bar{c})$ facts with a ϕ value greater than m_D, derivations that use these facts are delayed until some of the $p(\bar{d})$ facts that can be used along with the $q(\bar{c})$ facts can be discarded; this can improve the space utilization of the program. Note that if the facts with a ϕ index of m_D are also hidden, the set of locally saturated facts would not change, the value of m_D would not increase, and evaluation would not proceed any further. As seen in Example 7.1, hiding facts in this fashion could greatly reduce the space utilized by a program.

Our contribution in this section is twofold. First, we isolate the synchronization achieved by hiding facts in an evaluation from other components of space optimization methods. Second, Naughton and Ramakrishnan [1994] had the restriction that the *body_gap* be bounded above by a constant. We generalize this to handle the *body_gap* being bounded by an arbitrary function of facts.

7.2 Nested-Unit Synchronization

Programs that have been rewritten using Magic rewriting [Beeri and Ramakrishnan 1991] present opportunities for certain kinds of synchronization, which we present later. Familiarity with Magic rewriting is important for understanding the rest of this section.

Consider a Magic rewritten program P^{mg} obtained from a program P.[17] Let S be an SCC of P^{mg} such that S contains predicates from exactly two SCCs S_1 and S_2 of P, such that predicates defined in S_2 are used in S_1. Let $\mathscr{R}$ denote the rules in S. Let $\mathscr{R}_i$, $i = 1, 2$, denote the rules in $\mathscr{R}$ obtained by the Magic rewriting of rules in S_i. $\mathscr{R}_1$ can be partitioned into two sets of rules: $\mathscr{R}_1^{ext}$ containing the rules defining predicates of the form m_p, where p is defined in S_2, and $\mathscr{R}_1^{int}$ containing the rest of the rules in $\mathscr{R}_1$.

The Magic facts computed using the rules $\mathscr{R}_1^{ext}$ are referred to as *external* subgoals, in contrast to the Magic facts computed using the Magic rules in $\mathscr{R}_1^{int}$ and $\mathscr{R}_2$ which are referred to as *internal* subgoals. The Nested-Unit technique essentially views the rules in $\mathscr{R}_1^{ext}$ as generating subgoals, and solves them by obtaining the fixpoint of $\mathscr{R}_2$ augmented with these external subgoals. Nested-Unit synchronization should be used only if $\mathscr{R}_2$ is safely computable [Krishnamurthy et al. 1988].

```
Algorithm Nested-Unit_Synchronize (ℛ1, ℛ2)
  Let R1, ..., Rn be the rules in ℛ1^int.
  Let mR_{i,1}, ..., mR_{i,m_i} be the (Magic) rules in ℛ1^ext derived from
      rule R_i, in the left-to-right ordering of the literals in
      R_i from which they are derived.
```

[17] We assume a left-to-right sip order in the Magic rewriting of the program.

```
    Repeatedly apply the rules in R1^int, subject to the following re-
        strictions, until a fixpoint is reached.
    (1) Before applying a rule R_j from R1^int, do for k = 1...m_j
    (2) Apply mR_{j,k} and then compute a fixpoint of the rules R2.
end Nested-Unit_Synchronize.
```

PROPOSITION 7.2.1. *Consider* SCCs S_1 *and* S_2 *in a program with* $\mathscr{R}_1$ *and* $\mathscr{R}_2$ *defined as in the preceding. If* $\mathscr{R}_1$ *and* $\mathscr{R}_2$ *are evaluated using Nested-Unit synchronization, each predicate defined in* $\mathscr{R}_2$ *is base with respect to every rule in* $\mathscr{R}_1$.

PROOF. $\mathscr{R}_2$ is safely computable, therefore the correctness of the Magic sets transformation guarantees that *all* answers to subgoals generated using a rule in $\mathscr{R}_1^{ext}$ are computed before any rule uses any of the answers to the subgoals in a rule in $\mathscr{R}_1$. The external Magic rules corresponding to predicates in S_2 are applied in the left-to-right order, which is the sip-order used in the Magic rewriting, and no facts for predicates in S_1 are computed in this phase. An induction on the sip order then shows that the possible set of derivation steps using the rule applications in $\mathscr{R}_1$ would not change even if all the facts in the meaning of the program for predicates defined using S_2 were available. □

Although the preceding algorithm and proposition assume that S contains only predicates from two SCCs, S_1 and S_2, they can be extended to synchronize evaluation in the case where S contains predicates from multiple SCCs $S_1, \ldots, S_n$ of the original program. Multiple sets of rules $\mathscr{R}_i$, $1 \leq i \leq n$ are defined, and the synchronization technique ensures that if S_a defines a predicate used in S_b, $a \neq b$, then the evaluation of the rules in $\mathscr{R}_b$ will treat $\mathscr{R}_a$ in a nested fashion as before. The algorithm and proposition can also be extended in a straightforward manner to the case where S contains rules from multiple SCCs of a Magic rewritten program, rather than just a single SCC as was assumed earlier.

Several of the techniques for ensuring Conditions D or U used the notion of predicates being base with respect to rules. By using Nested-Unit synchronization we may enable the use of one of those techniques in a place where it may not otherwise be applicable.

Nested-Unit synchronization can be combined with the following straightforward technique for ensuring Condition U, which we call *Nested-Unit Discarding*. While computing the fixpoint of the rules in $\mathscr{R}_2$, discard facts, other than the external subgoals and answers that match the external subgoals, based on the restrictions of D and U to the rules of $\mathscr{R}_2$. Discard the external subgoals after computing the fixpoint of the rules in $\mathscr{R}_2$. Discard the answers to an external subgoal after applying the rule in $\mathscr{R}_1$ that generated the external subgoal.

The preceding technique may discard a fact before it has been used to make all the derivations that can be made using that fact. However, the use of Nested-Unit synchronization ensures that such a fact will be recomputed when required, and will be used to make any further required derivations.

Because derivations may be repeated, the resulting evaluation is not a semi-naive evaluation; however, it is a locally semi-naive evaluation.

7.3 Interleaved-Unit Synchronization

Interleaved-Unit synchronization is a form of synchronization that exploits the SCC structure of the program. 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 the evaluation proceeds SCC-by-SCC, the producer SCC evaluation must be completed before evaluation of the consumer SCCs can begin, and p facts must therefore be retained at least until the end of the evaluation 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 evaluation of SCCs, thereby making it possible to discard the p fact sooner, while retaining the advantages of an SCC-by-SCC semi-naive evaluation.

The preceding intuition can be extended to the case where the producer and the consumers can be units containing rules from multiple SCCs, rather than just one SCC. A unit of a program P is the *producer* for a predicate p if it contains all the rules from P that define p. A unit of a program P is a *consumer* for a predicate p if it is not the producer of p, and contains at least one body occurrence of p.

We present the technique by describing the interleaving of a producer unit (defining a single predicate p) and one or more consumer units for p. Any unit (other than the producer) that contains occurrences of p must be treated as a consumer and the producer and all consumer units must satisfy the following condition for the technique to be applicable.[18]

Condition Interleaved-Units.

—The producer and each of its consumer units must contain only monotonically increasing rules.

—The rules in the producer unit do not depend (directly or indirectly) on any predicates defined by rules in any of the consumer units.[19]

—In each consumer unit S_j, for each rule R that contains a body predicate occurrence p' of p, either (1) if there is an occurrence of a derived predicate in the body of R, then for each occurrence q' of any derived predicate q in the body of R, there exists 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'), $body\_gap(R', p', q') \leq \gamma_{p',q'}(q(\bar{b}))$; or (2) there is a bound $\max_{p'}$ such that for any fact $p(\bar{b})$ that can be used in the occurrence p', $\phi(p(\bar{b})) \leq \max_{p'}$.

[18]Although we consider only a single predicate defined in a producer unit and require that all consumers of the predicate satisfy Condition Interleaved-Units, it is possible to extend the condition as well as the synchronization technique to relax these restrictions.

[19]This condition is automatically satisfied if the units are SCCs of the program.

We now describe the *Interleaved-Unit* synchronization technique, which works on any subprogram that satisfies Condition Interleaved-Units. Consider a rule R in a consumer unit S_j. Let p' and q' be occurrences in the body of R of predicates p and q; let p be defined in a producer unit (of S_j) and q be derived with respect to R. We define the following indices:

$m(p', q')$
$= \max\{\{-\infty\} \cup \{\phi(q(\bar{b})) + \gamma_{p',q'}(q(\bar{b})) \mid q(\bar{b}) \text{ is an available fact}\}\}$

$M(p')$
$= \min\{m(p', q') \mid q' \text{ is a derived predicate occurrence in the body of } R\}$
$= max_{p'}$ if there is no derived predicate occurrence in the body of R

$\psi(p, S_j)$
$= \max\{M(p') \mid p' \text{ is an occurrence of } p \text{ in the body of any rule in } S_j\}$

$m(p', q')$ is the index of the largest (under the ϕ function) p fact that can possibly be used in p' with an available q fact in q'. $M(p')$ is the index of the largest p fact that can be used in the occurrence p' (with the set of currently known facts in S_j). The index of the largest p fact that can be used with the set of currently known facts in S_j is given by $\psi(p, S_j)$ and this index is available to the unit that defines p. Using these indices, Interleaved-Unit synchronization can be expressed as follows:

```
Algorithm Interleaved-Unit_Producer (S)
  (1)  repeat
  (2)      Let top = min_j{ψ(p, S_j) | S_j uses p and is waiting on S}.
  (3)      Evaluate S till no facts p(b̄) such that φ(p(b̄)) ≤ top can
               be derived.
             /* Tested using monotonicity; any technique may be
                    used to evaluate S. */
  (4)      Release any units S_j waiting on S such that ψ(p, S_j) = top.
  (5)  forever
end Interleaved-Unit_Producer

Algorithm Interleaved-Unit_Consumer (S_j)
  (1)  Evaluate S_j with the following restriction:
  (2)    Whenever new facts are made available for derived predi-
             cates in S_j do
  (3)      Update the indices m, M and ψ.
  (4)      Wait on producer units of S_j.
end Interleaved-Unit_Consumer
```

The previous description of the algorithm uses concurrent threads of execution for generality. It is straightforward to reformulate it, with a loss of concurrency and some extra checks, as a demand-driven sequential iteration. In this case, the evaluation of the consumer unit *invokes* the producer unit, rather than waiting on it; the evaluation of the producer units returns after computing all facts requested by a consumer.

Although the discussion so far assumed "monotonically increasing," if "increasing" is uniformly changed to "decreasing," the previous results and algorithms hold with simple modifications.

In the special case when all the consumer units of a given producer unit S_0 contain only nonrecursive rules using EDB predicates and predicates from

S_0, each of these consumer units is evaluated exactly once in an ordering determined by the ψ values of the consumer units. In such a situation, it is often beneficial to merge all the (nonrecursive) rules in the consumer units of S_0 with the rules in S_0. Further, these nonrecursive rules do not need to satisfy any bound $max_{p'}$ (as defined in Condition Interleaved-Units). Example 8.3.1 describes a program where this merging is very useful.

THEOREM 7.3.1. *If units $S_0, S_1, \ldots, S_m$ are evaluated using Interleaved-Unit synchronization with S_0 as the producer and $S_1, \ldots, S_m$ as its consumers, each predicate defined in S_0 is base with respect to every rule in $S_1, \ldots, S_m$.*

PROOF. Consider a single predicate p and a single consumer unit S_j that uses p. In order to prove the theorem, we need only show that $\psi(p, S_j)$ is indeed the largest ϕ value of any p fact that can be used in a derivation with any of the current set of derived facts in S_j. It then follows from the algorithm that any p fact that could possibly be used in a derivation step is indeed made available, and hence p is base with respect to every rule in S_j.

We show that $\psi(p, S_j)$ works as claimed by starting with $m(p', q')$. By the *body_gap* requirement of Condition Interleaved-Units and the definition of $m(p', q')$, $m(p', q')$ is indeed the index of the largest (under the ϕ function) p fact that can possibly be used in p' with an available q fact in q'. For a given rule R, if p facts with an index greater than some value n cannot be used in predicate occurrence p' with the available q facts for some predicate occurrence q', they cannot be used in a derivation step with the available facts for the derived predicates. Hence in the definition of $M(p')$ we take the minimum over all derived body predicate occurrences q'; $M(p')$ is then the index of the largest p fact that can be used in the occurrence p' with the set of currently available facts in S_j. In the definition of $\psi(p, S_j)$ we take the maximum over all predicate occurrences, therefore, $\psi(p, S_j)$ works as claimed. □

Example 7.3.2. Consider again the program from Example 6.3.4, and an SCC-by-SCC evaluation of the program.

$R1$: $fac\_list(0, [1])$.
$R2$: $fac\_list(N, [V \mid L]) :- N > 0, N < n, fac\_list(N - 1, L), fac(2 * N, V)$.
$R3$: $fac(0, 1)$.
$R4$: $fac(N, N * V) \quad :- N > 0, N < 2 * n, fac(N - 1, V)$.

This program has two SCCs, the lower one containing the predicate *fac* and the upper one containing *fac_list*. Let us call the lower SCC which is a producer of *fac* as $S1$ and the higher SCC, which is a consumer of *fac*, as $S2$. There is only one rule $R2$ in $S2$ that uses the predicate *fac*. This rule has a derived predicate *fac_list*. We assume that we use Basic Semi-Naive evaluation for the consumer SCC.

We derive the function γ that maps $fac\_list(N - 1, \_)$ to $2 * N - (N - 1)$, (and hence $fac\_list(N, \_)$ to $N + 2$) to bound $body\_gap(R2, fac(2 * N, V)$,

fac_list$(N-1, L)$). SCCs $S1$ and $S2$ satisfy Condition Interleaved-Units with this function γ that bounds body-gap. We can then use Interleaved–Unit evaluation to evaluate this program.

After each Basic Semi-Naive iteration of the consumer SCC (in Procedure Interleaved_Unit_Consumer) new facts are produced. Using these facts we find the maximum value of $\phi(fac\_list(N, \_)) + \gamma(fac\_list(N, \_))$. But this function simplifies to $2 * N + 2$. Thus if $fac\_list(n, \_)$ has been produced, we need *fac* facts with indices up to $2 * n + 2$. We then call Procedure Interleaved_Unit_Producer($S1$). SCC $S1$ then iterates, producing *fac* facts. Due to monotonicity of rules in $S1$, we know that when $fac(2 * n + 2, \_)$ has been produced, all *fac* facts with indices $\leq 2 * n + 2$ have been produced. Hence Procedure Interleaved_Unit_Producer returns, and Procedure Interleaved_Unit_Consumer continues with its next iteration.

Suppose we use Interleaved-Unit synchronization on this program, along with monotonicity to ensure Conditions D and U. The next question is, how much space is used? It is easy to see that in SCC $S2$, only two *fac_list* facts are retained at any point in the evaluation; each *fac_list* fact uses $O(n)$ space. As for SCC $S1$, we store at most facts with indices from $2*(n-1)$ to $2 * n$, which means at most 3 facts are stored. Thus we use a total of $O(n)$ space using this space optimization technique. If we do not discard any facts during the evaluation, we would use $O(n^2)$ space. By discarding facts during the evaluation, we have achieved an order of magnitude improvement in the space utilized in evaluating this program.

7.4 Using Inverted Rules

In several cases (such as monotonically increasing units that have been rewritten using the Magic Sets transformation), Condition Interleaved-Units is almost satisfied, except that the two units are monotonic in opposite directions. By using the notion of *inverted rules* introduced in Naughton and Ramakrishnan [1994], we can still use Interleaved-Unit evaluation in some cases.

Suppose that units S_1 and S_2 are monotonic in opposite directions, and rules in S_2 use predicates defined by rules in S_1. We can in some cases use the rules in S_1 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.

The intuition is to evaluate S_1 iteratively, discarding facts computed in S_1 based on Condition D and the restriction of Condition U to this unit. However, certain facts, that is, *fringe facts*, are retained during the evaluation of S_1. The "inverted" rules generated from S_1 are then evaluated (using the previously computed fringe facts) in an interleaved fashion with the consumer unit S_2. In general, the inverted rules may compute more facts than were computed earlier by the rules in S_1. However, computing a superset of the desired set of derived facts may be acceptable in some cases, for example, when the rules in S_1 compute magic facts; see Naughton and Ramakrishnan [1994] for a further discussion.

We present a generalized notion of inverted rules in Srivastava et al. [1994], and show how inverted rules could be used to ensure Condition U.

8. COMBINING TECHNIQUES

8.1 Sliding Window Tabulation

The Sliding Window Tabulation scheme of Naughton and Ramakrishnan [1994] is an example where the technique of adding inverted rules to a program is used in conjunction with delaying the first use of facts for synchronization and monotonicity of derivations and uses to ensure D and U. Sliding Window Tabulation works on programs that satisfy the following condition:

Condition Sliding_Window_Tabulation:[20]

(1) The Magic program P^{mg} has exactly two SCCs—the lower SCC S_2 only containing the Magic predicates (and rules defining them), and the higher SCC S_1 only containing the (derived) predicates (and the corresponding rules) of the original program.

(2) The rules in S_1 are monotonic in the opposite direction to the rules in S_2.

(3) The set of rules $\mathscr{R}_2$ in S_2 can be inverted to get $\mathscr{R}'_2$—the set of fringe facts being those Magic facts derived using $\mathscr{R}_2$ that do not generate any new Magic facts, and

(4) In P^{mg}, the body_gap in each rule with respect to each of the (nonMagic and corresponding Magic) predicates is bounded by a constant.

If the rewritten Magic program P^{mg} satisfies these conditions, the evaluation can be understood as follows:

```
Algorithm Sliding_Window_Tabulation_Eval (S1, S2)
  Let the set of inverted Magic rules obtained from the set of
     rules R2 in S2 be R'2.
  (1)  Evaluate the rules R2 using monotonicity to ensure Condi-
         tions D and the restriction of U to the uses of Magic
         facts in S2, while discarding facts; however, fringe
         facts are not discarded. The first use of facts is de-
         layed by hiding facts based on the body_gap of the
         (Magic) rules in S2.
  (2)  R'2 and the set of rules in S1 are evaluated using Inter-
         leaved- Unit Synchronization.[20] Monotonicity is used to
         ensure Conditions D and U and the first use of facts is
         again delayed by hiding facts. The ''lowest'' fact de-
         fined in S1 can be determined because R2 is evaluated be-
         fore S1.
end Sliding_Window_Tabulation_Eval
```

[20]Actually, we need to extend Interleaved Evaluation a little to handle the fully generality of Sliding Window Tabulation. Sliding Window Tabulation can handle some exit rules for which no bound $max_{p'}$ (defined in Condition Interleaved-Units) exists. It treats these rules as though they were derived rules, and makes only some Magic facts available to them at a time. Although Interleaved-Unit evaluation can be extended to handle such cases, we omit the tedious details of the extension.

Because $\mathscr{R}'_2$ is the set of inverted rules generated from $\mathscr{R}_2$, any facts discarded in Step (1) of the preceding algorithm are rederived when required in Step (2).

Using our generalized techniques for ensuring Conditions U, D and achieving synchronization based on monotonicity, the basic techniques of Sliding Window Tabulation can be extended in many ways beyond the class of programs described in Naughton and Ramakrishnan [1994]. One possible extension is based on synchronization of multiple consumer units with a single producer; another extension permits the body_gap of the rules to be bounded by some function of the facts, not just a constant.

8.2 A Framework for Combining Techniques

Recall that every space optimization method has three components—ensuring Condition U for facts before they are discarded, ensuring Condition D for facts before they are discarded, and synchronization techniques to ensure that as new facts get computed, others become eligible for discarding. We now discuss how these techniques (for synchronizing evaluation and for ensuring Conditions D and U for parts of a complex program) can be combined to obtain a space optimization method for the full program, and present a heuristic algorithm for this purpose.

The importance of our algorithm is twofold. First, it carefully incorporates the interactions between different space optimization methods in a modular fashion. There are several distinct ways to improve space utilization, but not all of them can be used on a given program in a consistent manner; the algorithm ensures that a consistent set of techniques is chosen. Second, the algorithm uses heuristics to prune the combinatorial explosion that would result in naively considering arbitrary combinations of methods.

8.2.1 *Orthogonality of Techniques.* The first point to note is that the choice of synchronization techniques affects the choice of the techniques for ensuring Conditions U and D—some techniques for ensuring U and D may be applicable only with certain synchronization techniques. For instance, Nested-Unit synchronization sets up subgoals when some facts are needed in a rule application; when the answers are computed (in a nested fashion) and used in the rule application, they automatically satisfy Condition U with respect to this predicate occurrence. This technique for ensuring Condition U, however, may not be applicable with other synchronization techniques. Further, because synchronization techniques determine which predicates can be treated as base (with respect to a rule or predicate) in an evaluation, they could affect the applicability of techniques (to ensure U and D) that depend on which predicates are base and which derived. This suggests that techniques for ensuring U and D for a subprogram be chosen after choosing a synchronization technique (for that subprogram).

The second point to note is that, given a synchronization strategy for a subprogram, the choice of a technique for ensuring U does not affect the correctness of a technique chosen for ensuring D, and vice versa. The applicability of techniques to ensure U for subprogram facts may depend on ensuring

D for (possibly other) subprogram facts; however, it does not depend on which techniques are used for this purpose.

The third point to note is that more than one technique may be applicable for ensuring D or U for a single literal (or rule), and a choice has to be made. The choice for one literal may affect the efficiency of the choice for another literal in that some of the overhead costs may potentially be shared by some combination of techniques but not by others.

In the process of obtaining a synchronization technique for an entire program, many choices have to be made, such as what synchronization techniques to use and what techniques to use for ensuring U or D for each literal and each rule. We do not address the issue of how to make an optimal choice in this paper, and leave it as an important open problem. However, in Section 8.2.2 we describe a heuristic for choosing synchronization techniques as well as techniques to ensure Conditions D and U, to obtain a space optimization method for the full program.

8.2.2 *A Heuristic for Combining Techniques.* In the following discussion we assume that we are given a program-query pair $\langle P, Q \rangle$. We expect Magic rewriting (or some variant thereof) to be used quite extensively in query optimization, and hence we describe how to obtain a space optimization method for the Magic rewritten form P^{mg} of program P rather than for P itself. We also assume that no rewriting is done on the program subsequent to Magic rewriting (although rewritings such as existential query optimization [Ramakrishnan et al. 1988] may be done prior to Magic rewriting). This assumption helps in presenting the algorithm concisely, but is not essential for the use of space optimization methods on the program.

We divide the program into units (i.e., subprograms), then determine the synchronization techniques to be used between units, and finally for each unit we choose the techniques to ensure U and D.

In order to describe the synchronization techniques to be used between units, we define a *unit graph* as follows: the units U_i form the nodes of the graph, and the edges are directed and are given labels from the following set: *Sequential*, *Nested*, *Interleaved*.[21] The edge labels specify how the program must be evaluated. If there is a Sequential edge from unit U_1 to unit U_2, then unit U_1 must be evaluated before the evaluation of unit U_2 is begun. The meaning of the other edges is defined similarly. This graph is required to be acyclic.

Later we describe a heuristic order in which to make the various choices for synchronization between units and techniques to ensure Conditions U and D for each unit. We then describe an algorithm that synchronizes the evaluation of a program based on the unit graph chosen. In the next section we present examples of the use of the heuristics.

Bottom-up evaluation of logic programs is typically performed using SCC-by-SCC BSN evaluation. Our heuristic for obtaining a space optimization method starts by initializing the units to be the SCCs of the program P^{mg},

[21] In Srivastava et al. [1994], we also consider edges labeled *Inverted*.

with *Sequential* edges between units. The evaluation of this unit graph corresponds to an SCC-by-SCC BSN evaluation of the program.

`Initialize`: Start with the SCCs $S_1, \ldots, S_m$ of P^{mg} as the initial units $U_1, \ldots, U_m$. Create a directed labeled edge $\langle U_i, U_j, \textit{Sequential} \rangle$, if a predicate defined in U_i is used in a rule in U_j.

`Create_Nested_Edges`: If a unit U_i (of P^{mg}) contains predicates from multiple SCCs of P, split U_i into units $U_{i1}, \ldots, U_{in_i}$ as described in Section 7.2. Each unit U_{ij} is nested within unit U_i. For each U_{ij}, if a predicate defined in U_{ij} is used in U_{ik}, $j \neq k$, create a directed labeled edge $\langle U_{ij}, U_{ik}, \textit{Nested} \rangle$.

`Create_Interleaved_Edges`: Consider a unit U such that all edges from U are labeled *Sequential*. If Condition Interleaved-Units is satisfied by U (as a producer unit) and all its consumers U_i, $1 \leq i \leq n$, relabel the edges from U to U_i as *Interleaved*. If all consumers of unit U have only nonrecursive rules, which use only EDB predicates and predicates defined in U, merge the consumer units into U, and collapse the nodes in the unit graph corresponding to the consumer units into the node corresponding to U. (Also change the edges in the unit graph to reflect this collapsing of nodes.)

`Create_Nested_Sub-Units`: The nested units created in the preceding steps are now reanalyzed by recursively applying the preceding steps independently to each nested unit treated as a program by itself.

`Decide_Hiding_Facts`: Analyze each unit (and sub-unit) U_i for the applicability of delaying the first use of facts during an evaluation of U_i, based on Condition Hiding_Facts.

`Analyze_UD_Applicability`: Check the applicability of all the techniques for ensuring U for each body predicate occurrence and for ensuring D for each rule in the resultant program.

`Choose_UD_Techniques`: Examine the set of applicable techniques for ensuring Conditions U and D and make suitable (heuristic) choices based on their relative "efficiency" and the overheads incurred. If a unit U has only *Nested* edges out of it, the rules in U use only EDB predicates and predicates defined in U, and no nontrivial technique for ensuring Condition D is applicable to predicates defined in U, use Nested-Unit Discarding.[22]

`Evaluate_Program`: We describe the evaluation of the program in a recursive fashion, starting from the unit U_q containing the rules defining the query predicate. Let U be the unit to be evaluated.

First consider the case when either U is U_q or the evaluation of U is called from a unit U' such that the label of the edge from U to U' is not *Interleaved*. Recursively evaluate all units U_i such that there is an edge labeled *Sequential* or *Inverted* from U_i to U. Next, if U has subunits, recursively evaluate each of the subunits of U that has no edges out of it. Next, if unit U has an edge labeled *Interleaved* into it, evaluate U using Algorithm `Interleaved-`

[22] Note that in this case, not ensuring Condition D for this unit does not adversely affect ensuring U for facts in other units.

Unit_Consumer in a demand-driven fashion, or else evaluate U iteratively. In either of these two cases, if there are units U_j such that the edges from U_j to U are labeled *Nested*, use Algorithm Nested-Unit_Synchronize to synchronize the evaluation of the U_js with the evaluation of U. Next consider the case when U is not U_q and the evaluation of U is called from a unit U' such that the label of the edge from U to U' is *Interleaved*. In this case, the evaluation of U is as described in the first case except that the evaluation of U proceeds until all facts required by U' have been computed, instead of computing until a fixpoint is reached.

We have described the synchronization for the various units in the program. Discarding of facts proceeds based on ensuring of Conditions U and D as described previously. This results in a space optimization method for the full program.

8.3 Obtaining a Space Optimization Method for an Example Program

The following program and query is typical of sequence querying in stock market applications (see, e.g., Roth et al. [1993]). In this domain, queries often require a scan over the entire dataset computing summary statistics. We demonstrate that a space-efficient evaluation can reduce the space required to evaluate this query from linear in the size of the database to a constant *independent* of the size of the database.

Example 8.3.1 (*N-Day Averages*). We are given a binary relation *sequence*(D, V), with the intended meaning that V is the value of the sequence on day D. We are interested in computing the average for each N day period beginning from a given day (this is indicated by the single fact in the *from*(D) relation); each period begins the day after the end of the previous period. The following program P_{avg} solves this problem. It defines the relation *ndayavg*(N, D, A), with the intended meaning that A is the N day average of the sequence beginning on day D.

$$
\begin{aligned}
&ndayavg(N, D, A) :- t1(N, D, N, V),\ A = V/N.\\
&t1(N, Day1, 1, V) :- from(Day1),\ sequence(Day1, V).\\
&t1(N, D2, 1, V2) \ :- t1(N, D, N, V1),\ D2 = D + N,\ sequence(D2, V2).\\
&t1(N, D, M, V) \quad :- M1 = M - 1,\ M1 < N,\ M1 > 0,\ t1(N, D, M1, V1),\\
&\qquad\qquad\qquad\qquad D2 = D + M1,\ sequence(D2, V2),\ V = V1 + V2.
\end{aligned}
$$

The Magic rewritten form of the preceding program for the query $?ndayavg(n, D, A)$ (where n is a constant) is as follows:

$$
\begin{aligned}
&R1:\ m\_t1(N, N) \qquad\quad :- m\_ndayavg(N).\\
&R2:\ ndayavg(N, D, A) \quad :- m\_ndayavg(N), t1(N, D, N, V),\ A = V/N.\\
&R3:\ t1(N, Day1, 1, V) \quad :- m\_t1(N, 1),\ from(Day1),\ sequence(Day1, V).\\
&R4:\ m\_t1(N, N) \qquad\quad :- m\_t1(N, 1).\\
&R5:\ t1(N, D2, 1, V2) \quad :- t1(N, D, N, V1),\ D2 = D + N,\ sequence(D2, V2).\\
&R6:\ m\_t1(N, M1) \qquad\quad :- m\_t1(N, M),\ M1 = M - 1,\ M1 < N,\ M1 > 0.
\end{aligned}
$$

$R7$: $t1(N, D, M, V)$ $:- m\_t1(N, M), M1 = M - 1, M1 < N, M1 > 0,$
$t1(N, D, M1, V1), D2 = D + M1,$
$sequence(D2, V2), V = V1 + V2.$

$R8$: $m\_ndayavg(n)$.

Choosing Synchronization Techniques. The unit structure (and edge labels) obtained using the method outlined in Section 8.2.2 is shown in Figure 1. Each unit in the figure indicates the predicates and rules it contains. All the edges in the unit graph are labeled *Sequential*. Though Condition Hiding_Facts is satisfied by unit U_3, each rule in this unit is linear, and hence delaying first use of facts is not useful.

Choice of Techniques for Ensuring Condition D. The rule in U_1 computes a single fact, and the fact is not discarded during the evaluation. Similarly, the rules defining m_t1 do not satisfy monotonicity or duplicate-freedom, and m_t1 facts are not discarded during the evaluation. The rules in U_3 are monotonically increasing, with the function ϕ mapping $t1(N, D, M, V)$ to $D + M$, and $ndayavg(N, D, A)$ to $N + D$; this is the only ϕ function, obtained by summing up the sizes of a subset of the arguments, that ensures monotonicity of the rules in U_3. Hence, Condition Monotonicity_D can be used to ensure Condition D for facts computed in U_3. The min-head-gap bounding function for the predicate *ndavavg* is 0, and for the predicate $t1$ is 1.

Choice of Techniques for Ensuring Condition U. Facts computed in U_1 and U_2 are not discarded. The rules defining $t1$ and *ndayavg* (in U_3) are linear, and Bounds_U applies trivially. The rules are also monotonically increasing, and Condition Monotonicity_U can also be used to ensure Condition U. Bounds_U is easier to test, therefore we use it.

Evaluation. Given the choice of synchronization techniques and techniques to ensure Conditions U and D, P_{avg}^{mg} is evaluated as follows. The evaluation starts with U_3 (which contains rules defining the query predicate), which first recursively invokes the evaluation of U_2; this in turn first recursively invokes the evaluation of U_1. Now the rules in U_1 are iteratively evaluated; no facts are discarded. Then the rules in U_2 are evaluated; again, no facts are discarded. Finally, the rules in U_3 are evaluated. During this evaluation, $t1$ facts are discarded based on Conditions Monotonicity_D and Bounds_U. It turns out that $t1$ facts are discarded at the end of the iteration following the iteration in which they are derived; the monotonicity ensures that these facts will not be derived again, and linearity of the rules ensures that they will not be used again. Similarly, *ndayavg* facts are returned to the user as they are computed, and discarded; monotonicity ensures that these facts will not be derived again. Facts for *ndayavg* are not used in the program, and hence satisfy Condition U trivially.

Note that the space optimization method for this program does not discard facts for all predicates defined in the program, unlike in previous examples.

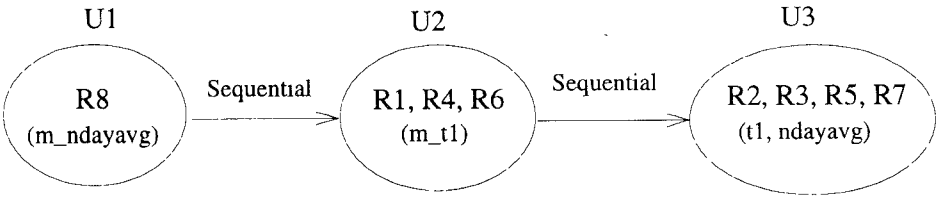

Fig. 1. Unit structure for P_{avg}^{mg}.

Improvements in Space Complexity. Given a query $?ndayavg(n, D, A)$, on a sequence database of size s, and a single fact $from(1)$, this evaluation stores a total of $n + 4$ nonEDB facts (1 fact for $m\_ndayavg$, n facts for m_t1 of the form $m\_t1(n, i)$, $1 \leq i \leq n$, 1 fact for $ndayavg$, and 2 facts for $t1$ computed in successive iterations). Note that the total space utilized in storing the nonEDB facts is *independent* of the size of the *sequence* database. If the space optimization methods described were not used, the total number of nonEDB facts computed would be $s + n + \lfloor s/n \rfloor + 1$, which is proportional to the size of the sequence database.

Note that Sliding Window Tabulation [Naughton and Ramakrishnan 1994] is not applicable in this example.

9. OVERHEADS

There are three aspects to the overheads involved with these 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 of the tests, and in other cases, at most 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 quite efficient, in the absence of function symbols. 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 6.3.3) 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 with 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.

10. CONCLUSION

In this paper we have described how to reduce the space required during bottom-up evaluation of logic programs and recursively defined views on databases by discarding facts. We showed that any space optimization method that discards facts during the evaluation has these 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 method for the full program. Because Sliding Window Tabulation [Naughton and Ramakrishnan 1994] can be shown to be just one way of combining techniques for each of these three components, our results subsume those in Naughton and Ramakrishnan [1994]. We presented a variety of techniques to ensure Conditions D and U. These are, of course, not exhaustive, and other useful techniques may be discovered, such as the one mentioned in Example 5.2.3.

An important direction of research is to define a set of evaluation primitives that can be implemented easily, and in turn can be used to implement the space optimization methods described in this paper. Future work also includes finding more techniques for ensuring each of the three components of an effective space optimization method. 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.

Finally, another direction of research is space optimization in active databases with complex rules, and in temporal databases, where facts that are "old" may no longer be required to support a predefined set of queries.

ACKNOWLEDGMENTS

We would like to thank Catriel Beeri, Won Kim, and the anonymous referees for suggestions that improved the content as well as the presentation of the paper.

REFERENCES

BALBIN, I., AND RAMAMOHANARAO, K. 1987. A generalization of the differential approach to recursive query evaluation. *J. Logic Program. 4*, 3 (Sept.), 259–262.

BANCILHON, F. 1985. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems—Integrating Database and AI Systems*, Brodie and Mylopoulos, Eds., Springer-Verlag, New York.

BEERI, C., AND RAMAKRISHNAN, R. 1991. On the power of Magic, *J. Logic Program. 10*, 3 & 4, 255–300.

DEBRAY, S. K., AND WARREN, D. S. 1989. Functional compositions in logic programs. *Trans. Program. Lang. 11*, 3 (July), 451–481.

GEHANI, N. H., JAGADISH, H. V., AND SHMUELI, O. 1992. Composite event specification in active databases: Model & implementation. *Proceedings of the 18th International Conference on Very Large Databases* (Vancouver, B.C., Aug.), 327–338.

HIRSCHBERG, D. S. 1975. A linear space algorithm for computing maximal common subsequences. *Commun. ACM 18*, 6 (June), 341–343.

JAGADISH, H. V., MUMICK, I. S., AND SHMUELI, O. 1992. Events with attributes in an active database. Tech. Rep. AT & T Bell Laboratories.

KEMP, D., RAMAMOHANARAO, K., AND SOMOGYI, Z. 1990. Right-, left-, and multi-linear rule transformations that maintain context information. In *Proceedings of the International Conference on Very Large Databases* (Brisbane, Australia), 380–391.

KRISHNAMURTHY, R., RAMAKRISHNAN, R., AND SHMUELI, O. 1988. A framework for testing safety and effective computability of extended Datalog. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Chicago, IL, May), 154–163.

LLOYD, J. W. 1987. *Foundations of Logic Programming*. 2nd ed., Springer-Verlag, New York.

MAHER, M. J., AND RAMAKRISHNAN, R. 1989. *Déjá vu* in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming* (Cleveland, OH), 963–980.

NAUGHTON, J. F., AND RAMAKRISHNAN, R. 1994. How to forget the past without repeating it. *J. ACM 41*, 6, 1151–1177.

NAUGHTON, J. F., RAMAKRISHNAN, R., SAGIV, Y., AND ULLMAN, J. D. 1989. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Portland, OR, May), 235–242.

RAMAKRISHNAN, R. 1988. Magic templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming* (Seattle, WA, Aug.), 140–159.

RAMAKRISHNAN, R., BEERI, C., AND KRISHNAMURTHY, R. 1988. Optimizing existential Datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Austin, TX, March), 89–102.

RAMAKRISHNAN, R., SRIVASTAVA, D., AND SUDARSHAN, S. 1994. Rule ordering in bottom-up fixpoint evaluation of logic programs. *IEEE Trans. Knowl. Data Eng.* (August). (A preliminary version appeared in *VLDB*, 1990).

ROTH, W. G., RAMAKRISHNAN, R., AND SESHADRI, P. 1993. MIMSY: A system for analyzing time series data in the stock market domain. In *Proceedings of the Workshop on Programming with Logic Databases* (Vancouver, B.C.), R. Ramakrishnan, Ed., 33–43.

SESHADRI, P., LIVNY, M., AND RAMAKRISHNAN, R. 1994. Sequence query processing. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. (Minneapolis, MN), 430–441.

SHMUELI, O. 1987. Decidability and expressiveness aspects of logic queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 237–249.

SRIVASTAVA, D., SUDARSHAN, S., RAMAKRISHNAN, R., AND NAUGHTON, J. 1991. Space optimization in deductive databases. Tech. Rep. 1063, University of Wisconsin, Madison.

TSUR, S., OLKEN, F., AND NAOR, D. 1990. Deductive databases for genomic mapping. Tech. Rep. TR-CS-90-14 (*Proceedings of the NACLP 93 Workshop on Deductive Databases*), Kansas State University.

ULLMAN, J. D. 1989. *Principles of Database and Knowledge-Base Systems, Volumes I and II*. Computer Science Press, New York.

Received June 1991; revised June 1995; accepted August 1995