How to Forget the Past Without Repeating It

Jeffrey F. Naughton* and Raghu Ramakrishnan†

Computer Sciences Department
University of Wisconsin-Madison, WI 53706, U.S.A.

Those who cannot remember the past are
condemned to repeat it.
— George Santayana

Abstract

Bottom-up evaluation of deductive database programs has
the advantage that it avoids repeated computation by storing
all intermediate results and replacing recomputation by table
lookup. However, in general, storing all intermediate results
for the duration of a computation wastes space. In this paper
we propose an evaluation scheme that avoids recomputation,
yet under fairly general conditions at any given time stores
only a small subset of the facts generated. The results consti-
tute a significant first step in compile-time garbage collection
for bottom-up evaluation of deductive database programs.

1 Introduction

A fundamental advance in deductive database technology has
been the invention of bottom-up query evaluation strategies
that retain the “focussing” properties of top-down evaluation
strategies. One of the key advantages of these bottom-up
strategies over common top-down approaches (such as Pro-
log) is that by storing all intermediate results, bottom-up
evaluation is able to replace the recomputation of a fact by
a simple lookup. In many cases this makes bottom-up eval-
uation polynomial time when the corresponding top-down
computation is exponential time. However, this efficiency in
the time domain is often accompanied by inefficiency in the
space domain.

In this paper we propose an evaluation scheme, called Slid-
ing Window Tabulation, that retains the time efficiency of
bottom-up computation while improving its space efficiency.
Intuitively, Sliding Window Tabulation stores a fact as long
as the fact could still be used to produce answers or avoid
redundant computation, but no longer. The need for such
an evaluation scheme is perhaps best demonstrated by an
element.

Example 1.1 Consider the problem of finding the longest
common subsequence in a pair of strings. This problem is
significant because it is paradigmatic 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.

The standard algorithm of Hirchberg [Hir75] for the
longest common subsequence can be expressed simply and
elegantly using Prolog notation (the program appears in Sec-
tion 5 of this paper). Unfortunately, on strings of length n,
the time complexity of Prolog is both $O\left(\binom{n}{2}\right)$ and $\Omega\left(\binom{n}{2}\right)$.
The function $\binom{n}{2}$ is exponential in $n$ and grows extremely
quickly. For example, if $n = 20$, we have that $\binom{n}{2} > 275 \times 10^5$; if $n = 100$, we have $\binom{n}{2} > 1.8 \times 10^{59}$. 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
Templates [Ram88] followed by Seminaive bottom-up evalu-
ation [Ban85], the running time is reduced to $O(n^3)$. This
is a dramatic improvement; unfortunately, the space require-
ment is also $O(n^2)$. In DNA sequence analysis, comparison
of strings of over $10^6$ 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 of
this size, the standard bottom-up approach will require over
a terabyte ($10^{12}$ bytes) of storage.

The evaluation algorithm presented in this paper, “Sliding
Window Tabulation,” evaluates the LCS program in $O(n^2)$
time and $O(n)$ space. To our knowledge this is the first
evaluation algorithm for deductive database queries that can
feasibly be applied to evaluate queries on this program over
large databases. □
The tradeoffs between memoing, recomputation, and efficient space management have been explored in the functional programming literature [Bir80, Coh83, Hil76], but to our knowledge have never been explored in the context of bottom-up evaluation of logic programs. A main contribution of this paper is to identify the problem of improving memory utilization in bottom-up evaluation of logic programs.

Sliding Window Tabulation, presented in Section 4, differs significantly from techniques suggested for functional programs in [Coh83, Bir80], and presents an interesting contrast: Rather than explore program schemas and schema transformations, we develop a uniform approach based on optimizing the structure of programs generated by the Magic Templates algorithm.

Our algorithm for tabulation of a program and query consists of three phases: 1) First, the program is analyzed to determine if tabulation can be profitably applied. 2) Next, the program is rewritten using the Magic Templates rewriting algorithm. 3) Finally, the resulting rewritten program is evaluated by an algorithm that attempts to store facts only as long as necessary, using information derived from the analysis of the program done in step 1.

The remainder of this paper is organized as follows. We give a brief overview of bottom-up evaluation in Section 2. We give an overview of Sliding Window Tabulation in Section 3, and consider the algorithm in more detail in Section 4. We present testable conditions for applicability of this algorithm and several results about its performance. We present a detailed discussion of the LCS problem introduced in Example 1.1 in Section 5. We present conclusions and directions for future work in Section 6.

2 The Bottom-Up Approach

To provide context for the discussion of Sliding Window Tabulation, and to make this paper self-contained, in this section we present an overview of the bottom-up approach to deductive database query evaluation. This approach has been developed without consideration for space utilization.

The first step in bottom-up evaluation is to rewrite the program using the Magic Templates transformation [Ram88]. Magic Templates extends the Magic Sets rewriting algorithm [BMSU86, BR87] to deal with non-ground facts. The important properties of the algorithm for the purposes of this paper is that given a program $P$ and a query $q$, Magic Templates produces a new program $P_{m9}$ such that evaluating $P_{m9}$ bottom-up produces no irrelevant facts.

After applying the Magic Templates transformation, the resulting program $P_{m9}$ is evaluated bottom-up using a Seminaive algorithm. Seminaive fixpoint evaluation [Ban85] ensures that derivations are not repeated in subsequent iterations, by considering in each iteration only rule instantiations that utilize at least one new fact generated in the previous iteration. (This is considered in more detail in Section 4.2.)

We now give a brief description of the Magic Templates algorithm. For details, consult [Ram88]. The Magic Templates algorithm [BMSU86, BR87, Ram88] transforms a program, for a given query form, in such a way that the Seminaive evaluation of the transformed program generates no irrelevant facts. The initial rewriting of a program and query is guided by a choice of sideways information passing strategies, or sips. For each rule, the associated sip is the (partial) order in which the body literals are to be evaluated.

The idea is to compute a set of auxiliary predicates that contain the goals. The rules in the program are then modified by attaching additional literals that act as filters and prevent the rule from generating irrelevant tuples. As a first step, however, we produce an adorned program in which predicates are adorned with an annotation that indicates which arguments are bound and which are free. Adornments for the program are determined from the query and the choice of sips. The Magic Templates algorithm is a two-step transformation in which we first obtain the adorned version of $P$ and then apply the following transformation:

**Definition 2.1** [The Magic Transformation] We construct a new program $P_{m9}$. Initially, $P_{m9}$ is empty.

1. Create a new predicate $magic_p$ for each predicate $p$ in $P$. The arity is that of $p$.
2. For each rule in $P$, add the *modified version* of the rule to $P_{m9}$. If rule $r$ has head, say, $p(f_1)$, the modified version is obtained by adding the literal $magic_p(f_1)$ to the body.
3. For each rule $r$ in $P$ with head, say, $p(f_1)$, and for each literal $q_i(f_i)$ in its body, add a *magic rule* to $P_{m9}$. The head is $magic_q(q_i)$. The body contains all literals that precede $q_i$ in the sip associated with this rule, and the literal $magic_p(f_1)$.
4. Create a *seed fact* $magic_q(\epsilon)$ from the query.

A simple optimization is to delete all argument positions corresponding to free arguments from the magic predicates, since these positions always contain distinct variables. In the sequel, we will refer to the above two step transformation with this optimization as the Magic Templates algorithm. We now illustrate the bottom-up approach on a program to compute the Fibonacci numbers.

**Example 2.1** The following program computes the Fibonacci numbers.

\[
\begin{align*}
fib(0, 1) & . \\
fib(1, 1) & . \\
fib(N, X1 + X2) & :- N > 1, fib(N - 1, X1), fib(N - 2, X2).
\end{align*}
\]

Magic Templates applied to this program and the query $fib(n, X)$ produces the "magic" rules

\[
\begin{align*}
m._{fib}(n) & . \\
m._{fib}(N - 1) & :- N > 1, m._{fib}(N) . \\
m._{fib}(N - 2) & :- N > 1, m._{fib}(N) .
\end{align*}
\]

and the modified original rules

279
fib(0,1) :- m_fib(0).
fib(1,1) :- m_fib(1).
fib(N, X1 + X2) :- m_fib(N), N > 1, fib(N - 1, X1), fib(N - 2, X2).

One may verify that this program, when evaluated using Seminaive bottom-up, computes the answer to the query in linear time. □

3 Overview of Sliding Window Tabulation

Sliding window tabulation of a program/query pair \((P, q)\) consists of two phases. In phase one, the magic rules of \(P^{mg}\) are repeatedly applied in order to determine the set of all basis facts relevant to the query. In phase two, the modified program rules of \(P^{mg}\) and the inverted magic rules of \(P^{mg}\) (inverted magic rules are defined in Definition 3.3 of Section 4) are repeatedly applied to work back up towards the query goal in order to generate answers. Only relevant basis facts may be used in this second phase of sliding window tabulation. Often we will refer to phase one as the “down” phase, since phase one works “downwards” toward basis facts. Similarly, we will refer to phase two as the “up” phase, since it works “upwards” toward answers.

Both the “down” phase and the “up” phase are performed using a modification of Seminaive evaluation. The main idea behind this modification is to partition the facts produced during the fixpoint into possibly overlapping sets called windows. We emphasize that the partitioning of facts into windows is conceptual; as detailed in Section 4, in practice the algorithm accomplishes the partitioning by deriving a “windowing” function that maps facts to windows as they are produced during the query evaluation.

The computation of the answer to \((P, q)\) begins with the window containing the “seed” magic fact, and proceeds by sliding the window downward, using the magic rules of \(P^{mg}\) to compute new magic facts as it goes, deleting the facts in a given window after they have been processed. The downward phase terminates when the evaluation of the window containing the “lowest” magic fact reachable from the query goal has been completed.

The upward phase starts with the lowest window that could contain a relevant basis fact, and proceeds by sliding the window upwards, computing program facts by applying the original rules and inverted magic rules of \(P\) along the way, again deleting the facts from a given window after it has been processed. The upward computation terminates when the current window has slid up to the window containing the highest answer to the query goal.

The key property of Sliding Window Tabulation with respect to efficient space utilization is that 1) during the “down” phase, no magic fact \(m\) is stored after the current window has slid below the lowest window that contains \(m\), and 2) during the “up” phase, no program fact \(f\) or magic fact \(m\) is stored after the current window has slid above the uppermost window that contains \(f\).

A variant of Sliding Window Tabulation stores all magic facts encountered in the down phase, and uses these stored magic facts instead of inverted magic rules in the up phase. This requires more storage than basic Sliding Window Tabulation, but can be more time-efficient if basic sliding window “overtabulates” on \((P, q)\). (Overtabulation is described in Subsection 4.4.)

We will use the program to compute Fibonacci numbers, given in Example 2.1, as a running example. While the program is simple, it illustrates some important aspects of sliding window tabulation. A more complex example of sliding window tabulation is given in Section 5.

3.1 Detecting Applicability

In this subsection we develop sufficient conditions for Sliding Window Tabulation to apply, and give an algorithm that tests for these conditions.

A useful property of many programs is that the facts for magic predicates in \(P^{mg}\) can be computed independent of the facts for derived predicates of the original program. This allows us to separate the evaluation of \(P^{mg}\) into two phases: first, compute magic facts, and then compute program facts by applying the modified rules. To formalize this idea, we use the following definition:

Definition 3.1 Define the relation \(p \rightarrow q\) such \(p \rightarrow q\) holds if there is a rule with a \(q\) literal in the head and a \(p\) literal in the body. We let \(\rightarrow\) denote the transitive closure of \(\rightarrow\). □

Intuitively, if \(p \not\rightarrow q\), then \(p\) is used to define \(q\).

Definition 3.2 Let \((P, q)\) be a program-query pair, and let \(P^{mg}\) be the result of the Magic Templates rewriting of \((P, q)\). Then \((P, q)\) has the sliding window property if:

1. If \(p_m\) is a derived predicate in \(P^{mg}\), and \(m.p_n\) is a magic predicate, then \(p_m \rightarrow m.p_n\) does not hold.

2. There are constants \(u\) and \(l\), where \(l \leq u\), and a set of windows \(\mathcal{W}_l\), where \(l \leq i \leq u\), such that the following two conditions hold:

   (a) Consider a rule instantiation \(p := m.p_1, p_2, \ldots, p_k\) in the fixpoint evaluation of \(P^{mg}\). Suppose that \(p\) appears in window \(\mathcal{W}_j\). Then there must be a window \(\mathcal{W}_i\), where \(j \leq i \leq u\), such that the following two conditions hold:

   (1) Consider a rule instantiation \(p := m.p_1, p_2, \ldots, p_k\) in the fixpoint evaluation of \(P^{mg}\). Suppose that \(p\) appears in window \(\mathcal{W}_j\). Then there must be a window, say \(\mathcal{W}_i\), where \(j \geq i\), that contains every fact that appears in the body.

   (b) Consider an instantiation of a magic rule \(m.p_1 := m.p_1, p_2, \ldots, p_k\) in the fixpoint evaluation of \(P^{mg}\). Suppose that \(m.p_1\) appears in window \(\mathcal{W}_j\). Then there must be a window, say \(\mathcal{W}_i\), where \(j \geq i\), that contains every fact that appears in the body.

□

We now define inverted magic rules.

Definition 3.3 [Inverted Magic Rules] Let \(r\) be a magic rule of the form \(m.p := m.q, e_1, \ldots, e_n\), where \(m.p\) and \(m.q\) are magic predicates and the \(e_i\) are EDB predicates. Then the
The inverted magic rules for $P^{mg}$ consist of the inversion of each magic rule of $P^{mg}$. □

Example 3.1 For the Fibonacci program of Example 2.1, the inverted magic rules are

$$m_{-}fib(N) := N > 1, m_{-}fib(N - 1).$$

$$m_{-}fib(N) := N > 1, m_{-}fib(N - 2).$$

Note that the fact $m_{-}fib(n)$ does not have a corresponding inverted version. □

To discover whether or not a program has the sliding window property, we first need to provide a mechanism for determining to which window or set of windows a given fact or goal belongs. We will assume that EDB facts cannot be discarded, and thus consider these facts to be in every window. Recall that the EDB relations are by definition those relations that are defined by tuples stored in the system rather than defined by rules. A user might be justifiably upset if a query evaluation algorithm deleted these relations as a side effect of answering a query. For this reason, we can restrict our attention to the set of magic facts and derived program facts in determining windows.

We simplify the search for windows by defining the set $W_i$ in terms of the bound arguments of program facts and all arguments of magic facts. This uses the property that the set of bound arguments in the goals for a given predicate is exactly the set of bound arguments of the facts for that predicate whenever the bound arguments contain only ground terms.

We determine the windows $W_i$ by defining a function that maps goals and facts to integers. The intent is that a given magic rule $r_i$ and IDB predicate $p$ are joined by a path in the Rule/Goal graph. The size of such a path is the function that maps to the same integer. For notational convenience, we will often use the generic function $\phi$ to represent the goal to the magic predicate.

A high-level description of an algorithm to detect when a program has the sliding window property follows. The algorithm takes as input the program $P^{mg}$ corresponding to a program/query pair $(P, q)$. If the algorithm can verify that the program has the sliding window property, it returns "true". Otherwise, it returns "false".

Method: Construct the Rule/Goal graph for $P^{mg}$. Then, for each IDB predicate $p$, check that there is a path from the node for $p$ in this graph to the node for a magic predicate.

The running time of this procedure is $O(n^2)$, where $n$ is the number of predicates in $P^{mg}$.

Procedure 3.2 [Groundness]

Input: A program $P^{mg}$.

Output: True, if in $P^{mg}$, there are no pairs of IDB predicates $p, q$ such that $p \rightarrow m.q$ as defined in Definition 3.1; false otherwise.

Method: Construct the Rule/Goal graph for $P^{mg}$. For each EDB predicate $p$, check that there is no path from the node for $p$ in this graph to the node for a magic predicate.

The running time of this procedure is linear in the size of $P^{mg}$. Procedures 3.1 and 3.2 are independent of the particular class of candidate $\phi$ functions considered by Algorithm 3.1. By contrast, the remainder of the procedures in that algorithm are highly dependent on the chosen class of candidate $\phi$'s. We have presented the algorithm in generic form, to accommodate new classes of $\phi$'s as they are developed. In the remainder of this section, we give specific instances of the algorithms for an especially useful class of $\phi$'s, as discussed below.

The detection algorithm works by enumerating a set of possible choices for $\phi$ and checking for the sliding window property with each choice.
The class of candidate $\phi$ functions considered in this paper are all based upon the sum of the sizes of a subset of the bound arguments of predicates, or the additive inverse of this sum of sizes. To define the size of an argument, we divide the bound arguments into two types: those that contain integers, and those that contain structured terms.

We will use the convention that all terms are uninterpreted, with the exception of arithmetic expressions (as is standard in logic programming). With this convention, the "size" of an argument $a$, denoted $s(a)$, is defined as follows:

1. For an integer expression $e$: if $e$ appears in an argument position of a term (that is uninterpreted), then $s(e) = 1$, else $s(e) = e$. (For example, $s(5 + 6) = 1$ in the term $f(5 + 6)$, where $f$ is an uninterpreted function symbol.)
2. The size of a variable $X$ is unknown, and is represented by $s(X)$.
3. The size of an atom in a structured argument is 1.
4. The size of a term $f(t_1, t_2, \ldots, t_p)$ is defined by
   
   $$s(f(t_1, t_2, \ldots, t_p)) = 1 + \max(s(t_1), s(t_2), \ldots, s(t_p))$$

For example, suppose that $\phi$ is the sum of arguments one and three of $p$, and that argument one has been determined to be a structured argument while argument three is an integer argument. Then

$$\phi(p(f(g(X), 6 + 10), X, 4 + 2)) = s(g(h(X)), 5 + 10)) + s(4 + 2)$$
$$= 1 + \max(s(g(h(X))), s(6 + 10)) + 4 + 2$$
$$= 1 + \max(1 + s(h(X)), 2) + 6$$
$$= 1 + \max(2 + s(X), 2) + 6$$
$$= 8 + s(X)$$

Note that in this case the expression $6 + 10$ appears in a structured argument, hence it is uninterpreted, and has size 2. On the other hand, the expression $4 + 2$ appears in an integer argument, hence it evaluates to 6. Also, this example indicates that the result of applying $\phi$ to a term may involve a variable. The function $\phi$ computes the sum of the sizes of a subset of the bound arguments of its argument goal or predicate. For this class of $\phi$ functions, we can enumerate all such choices for $\phi$ by enumerating the subsets of the bound arguments.

Procedure 3.3 [NextPhi]

Input: A program $Pmg$.

Output: True, if there are alternatives for $\phi$ that have not been returned by any previous call to NextPhi. In this case the variable $\phi$ is set to the next such alternative. The procedure returns False otherwise.

Method: NextPhi considers only functions $\phi$ that are defined to be the sum of the sizes of a subset of bound arguments, or the negative of the sum of the sizes of a subset of bound arguments. NextPhi retains in static storage the subset of bound arguments it returned in the previous call, and whether it was returned with positive or negative sign; on each call it updates this static storage and returns either the additive inverse of the current subset or the next subset in some order of enumeration. If in previous calls all subsets have been returned with both positive and negative sign, NextPhi returns false.

Each call to NextPhi can be processed in time linear in the number of bound arguments. The total number of calls to NextPhi by Algorithm 3.1 is exponential in the number of bound arguments in a given predicate. However, since we expect the number of bound arguments to be small, an exhaustive algorithm is sufficient. (In the examples considered in this paper, at most two arguments are bound; this means that the number of cases to be considered is six, three with each sign.)

Next we turn to the procedures that check each candidate $\phi$ function to see if it satisfies the sliding window property.

Procedure 3.4 [CheckMonotonic]

Input: A program $Pmg$ and a candidate windowing function $\phi$.

Output: True, if for every possible instantiation of a rule in the bottom-up evaluation of $Pmg$, if $p$ is the predicate instance in the head of the rule, and $p_1, \ldots, p_k$ are the recursive predicate instances in the body of the rule,

$$\phi(p) \geq \max(\phi(p_1), \ldots, \phi(p_k))$$

must hold.

Method: For each modified original rule $r$ in $Pmg$,

1. Apply $\phi$ to each literal in the head and body of $r$, and simplify using the definition of size of an argument and standard arithmetic.
2. Determine the literal $p_i$ in the body of $r$ such that $\phi(p_i)$ is maximal over all predicates in the body.
3. Attempt to produce an arithmetic tautology by simplifying $\phi(p) \geq \max(\phi(p_1), \ldots, \phi(p_k))$ using standard arithmetic.

If the procedure succeeds on step 3 for all rules in $P$, return true, else return false.

Example 3.2 Consider again the Fibonacci program of Example 2.1. Since in the Fibonacci program there is only one bound argument, the choices for $\phi$ are just $\phi(fib(N, x)) = N$ or $\phi(fib(N, x)) = -N$. The following shows that Procedure CheckMonotonic succeeds with the first alternative.

To test for monotonicity, Procedure CheckMonotonic must compare the first instance of $fib$ in the body of the rule and the instance in the head. That is, it must test if $\phi(fib(N, X)) \geq \phi(fib(N - 1, X))$ for all possible instantiations of the two predicates. We have that

$$\phi(fib(N, X)) \geq \phi(fib(N - 1, X))$$

holds if and only if $s(N) \geq s(N - 1)$. Since this reduces to $0 \geq -1$, which holds unconditionally, the monotonicity constraint is satisfied. The monotonicity constraint for the second instance of $fib$ and the instance of $m-fib$ are similar.

For an example where CheckMonotonic returns with failure, consider the transitive closure.
and the query \( t(1,Y) \)? Here, again, there are only two choices for \( \phi \), the size of the first argument of \( t \), or the additive inverse of the size of the first argument. With the first choice, we will have to verify that there exists an \( h \) such that the difference 
\[
\max(\phi(t(X,W)), \phi(t(W,Y))) - \min(\phi(t(X,W)), \phi(t(W,Y)))
\]

is less than \( w \). Here we have 
\[
\max(\phi(t(X,W)), \phi(t(W,Y))) - \min(\phi(t(X,W)), \phi(t(W,Y)))
\]

reduces to \( \max(s(X), s(W)) - \min(s(X), s(W)) \). No more simplifications can be made, since nothing is known about \( X \) and \( W \), so the test fails. The case for the other choice of \( \phi \) is similar. 

Finally, we turn to finding limits on the windows involved.

**Procedure 3.6 [Find Limit]**

*Input:* A program \( P^{mg} \) and a candidate windowing function \( \phi \).

*Output:* True, if there are constants \( c_+ \) and \( c_- \) such that any answer \( a \) to \( q \) must have \( \phi(a) \geq c_+ \) and any magic fact \( m \) produced in the bottom-up evaluation of \( P^{mg} \) must have \( \phi(m) \leq c_- \). In this case return \( c_+ \) and \( c_- \). Otherwise return false.

*Method:* We consider the cases for \( c_+ \) and \( c_- \) separately. There are two cases to consider, depending on whether \( \phi(x) \geq 0 \) for all facts \( x \), or \( \phi(x) \leq 0 \) for all facts \( x \). One or the other must hold, since sizes are never negative and \( \phi \) is either a sum of sizes or the inverse of a sum of sizes.

First, consider the case where \( \phi \geq 0 \).

1. Since by definition every answer agrees with \( q \) on the bound arguments, we may always set \( c_+ \) to \( \phi(q) - h \).
2. The constant \( c_- \) is more complex. Suppose that \( \phi \) considers the arguments \( a_1, \ldots, a_m \). Then if argument \( a_i \) contains structured arguments, define \( l_i = 0 \). Otherwise, \( a_i \) must contain an integer argument. If the variable in \( a_i \) is \( X - k \), and \( X \) appears in the body of the rule in a predicate \( X \geq y \), then define \( l_i = y - k \). Define \( c_- = \sum l_i \). Verify by expansion that \( c_- \) works for all predicates in \( P \).

For the case where \( \phi \leq 0 \), the roles of \( c_+ \) and \( c_- \) are reversed. If Step 2 was successful, return \( c_- \) and \( c_+ \), otherwise return failure. 

**Example 3.4** Returning once more to the Fibonacci example, consider the magic rule 
\[
m-fib(N-2) :- N > 1, m-fib(N).
\]

and assume that we are considering \( \phi(fib(N,X)) = \phi(m-fib(N)) = N \). Then FindLimit must find some constant \( c_- \) such that 
\[
\phi(m-fib(N-2)) \geq c_- \n\]

In the body of the rule, we have that \( N > 1 \), which implies that \( N - 2 \geq 0 \), so this condition is satisfied for \( c_- = 0 \).
Furthermore, if the query is \( \text{fib}(n, X) \)?, since we have already determined that \( h = 2 \), we set \( c_u = n - 2 \).

Combining everything in Examples 4.2 through 4.4, we get that for the query \( \text{fib}(n, X) \)? on the fibonacci example, the relevant windows are

\[
\begin{align*}
W_0 &= \{ m \cdot \text{fib}(N) \land \text{fib}(N, X) \mid 0 \leq N \leq 2 \} \\
W_1 &= \{ m \cdot \text{fib}(N) \land \text{fib}(N, X) \mid 1 \leq N \leq 3 \} \\
& \quad \vdots \\
W_{n-3} &= \{ m \cdot \text{fib}(N) \land \text{fib}(N, X) \mid n - 3 \leq N \leq n - 1 \} \\
W_{n-2} &= \{ m \cdot \text{fib}(N) \land \text{fib}(N, X) \mid n - 2 \leq N \leq n \}
\end{align*}
\]

**Theorem 3.1** If Algorithm 3.1 returns successfully when invoked on \((P, q)\), then \((P, q)\) has the sliding window property.

### 4 Sliding Window Tabulation

In this section we consider Sliding Window Tabulation in more detail.

#### 4.1 A “Naive” Description

The focus in Sliding Window Tabulation is on how we can discard facts as early as possible. An orthogonal concern is how to avoid repeating the same inferences. The Seminaive bottom-up evaluation algorithm can be adapted to ensure that Sliding Window Tabulation does not repeat any inferences. We consider this adaptation in the next subsection. (We consider the adaptation, or some equivalent technique for avoiding repeated inferences, to be an integral part of Sliding Window Tabulation. We have presented the ideas separately for ease of exposition.)

Sliding Window Tabulation of a (rewritten) program \( P^{mg} \) proceeds in two phases. In phase one, the “down” phase, only the magic rules are applied. Initially, the only magic fact is the query, which is in the highest window. For each window, processing consists of repeatedly applying the magic rules until no new facts can be derived. The important constraint is that in applying a magic rule, only facts in the current window can be used to instantiate the body. The Monotonicity condition in the definition of the sliding window property ensures that generated facts belong to either the current window or to some higher window; those in higher windows are saved for processing later. After a window is processed, we discard all facts in this window that do not belong in subsequent windows also. The “up” phase terminates when we have processed the highest window; all answers to the query are contained in the facts that belong to this window.

#### 4.2 A “Seminaive” Formulation

In this subsection we describe how Seminaive evaluation [Ban85] can be adapted to Sliding Window Tabulation.

##### 4.2.1 Seminaive Evaluation

We present a brief overview of Seminaive evaluation. Seminaive evaluation works by identifying “differentials,” which are new predicates that contain tuples produced in the last iteration. Consider a program that is to be evaluated using Seminaive evaluation. The program is first rewritten in order to define the new “differential” predicates. Suppose the program contains the following rule:

\[
P := p_1, p_2, \ldots, p_k, q_1, q_2, \ldots, q_m.
\]

Let the \( p_i \)'s be derived predicates and the \( q_i \)'s be EDB predicates. A set of rewritten rules is generated from this rule, each of the form \( \text{rew} : \text{term} = q_1, \ldots, q_m \). There is one such rewritten rule for each term in the expansion of \( p_1^\text{old} \land p_2^\text{old} \land \cdots \land p_m^\text{old} \). In evaluating the program, in each iteration each of the seminaive rules is applied, followed by updating the relations as follows:

\[
\begin{align*}
p_1^\text{old} &:= p_1^\text{old} + \delta p_1^\text{new} \\
\delta p_1^\text{old} &:= 0 \\
\delta p_1^\text{new} &:= 0.
\end{align*}
\]

The iteration continues until all the relations \( \delta p_i^\text{old} \) are empty.

#### 4.3 Sliding Window

For convenience, we introduce the function \( \Phi_w \), where for a set of facts \( S \), we define \( \Phi_w(S) \) to be the subset of \( S \) that is contained in window \( w \). We also define \( \text{unused}(S) \) to be all facts in \( S \) that were never used in any instantiation of a rule. Recall that during the “down” phase of Sliding Window Tabulation, only the magic rules are applied. In the “up” phase on the other hand, only the modified program rules
and the inverted magic rules are applied. Sliding Window Tabulation differs from Seminaive in the updating phase following each iteration, both in the down and up phases.

Also, for each magic predicate $m.p_i$, we introduce the predicates $m.p_i^{new}$ and $m.p_i^{fringe}$ (in addition to the predicates introduced by the standard seminaive rewriting.) Intuitively, $m.p_i^{new}$ stores magic facts that belong to a window other than the window currently being processed; $m.p_i^{fringe}$ stores the fringe magic facts encountered during the "down" phase. As is discussed below, there is no need for $p_i^{new}$ or $p_i^{fringe}$ for program predicates $p_i$.

Consider first the down phase. The initialization is simple — the current window is set to the highest window, $W_l$, and all relations are empty, with the exception that $\delta m_q^{old} = seed$, where $q$ is the query predicate and $seed$ is the magic fact corresponding to the query.

As with the standard Seminaive evaluation, all magic rules are applied in every iteration. However, instead of performing the Seminaive updates at the end of an iteration in the processing of window $w$, perform the following assignments for each magic predicate $m.p_i$:

1. $m.p_i^{new} := m.p_i^{new} + \delta m.p_i^{new} - \Phi_w(\delta m.p_i^{new})$;
2. $m.p_i^{old} := m.p_i^{old} + \delta m.p_i^{old}$;
3. $\delta m.p_i := \Phi_w(m.p_i^{new}) - m.p_i^{old}$;
4. $m.p_i^{new} := \emptyset$;

Steps 1) just saves facts in lower windows for later processing. Steps 2) - 4) are the usual seminaive updates, except that Step 3) only retains facts in the current window from $\delta m.p_i^{new}$ (recall that the remaining facts are saved for later processing, in Step 1)).

The processing of a window continues until all relations $\delta m.p_i^{old}$ are empty. Next, between the processing of windows $w$ and $w-1$, unless of course $w$ is the lowest window, the following updates must be performed:

5. $\delta m.p_i^{old} := \Phi_{w-1}(m.p_i^{new})$;
6. $m.p_i^{old} := \Phi_{w-1}(m.p_i^{old})$;
7. $m.p_i^{fringe} := m.p_i^{fringe} + unused(\Phi_w(m.p_i^{old}) - \Phi_{w-1}(m.p_i^{old}))$;

Steps 5) - 6) initialize the processing of the next window ($w-1$). Note that Step 6) just discards some facts from window $w$ that have been processed and are no longer needed. Step 7) saves "fringe" facts, to be used later in the up phase.

The down phase terminates after the processing of the lowest window has been completed. At this point, we perform the update

8. $m.p_i^{fringe} := unused(m.p_i^{old})$;

for each magic predicate.

To initialize the "up" phase, we begin with the updates

9. $m.p_i^{new} := m.p_i^{fringe}$;
10. $\delta m.p_i^{old} := \Phi_{w+1}(m.p_i^{new})$;

In the up phase, we fire the program rules and the inverted magic rules. The updates to the magic predicates after each iteration are identical to those of the down phase. However, the updates to the program predicates are simplified, for the following reason: for any program predicate $p_i$, the predicate $p_i^{new}$ is uniformly empty. This follows because every original program rules with head $p_i$ is "guarded" by the magic predicate $m.p_i^{old}$, which at all times contains only facts in the current window. Hence the updates for program predicates are just the usual seminaive updates, repeated here for convenience:

11. $p_i^{old} := p_i^{old} + \delta p_i^{old}$;
12. $\delta p_i^{new} := \delta p_i^{new} - p_i^{old}$;
13. $p_i^{new} := \emptyset$;

In between the processing of windows $w$ and $w+1$ on the up phase, where $w$ is not the top window, the following updates must be performed:

14. $\delta m.p_i^{old} := \Phi_{w+1}(m.p_i^{new})$;
15. $m.p_i^{new} := m.p_i^{new} - \Phi_{w+1}(m.p_i^{new})$;
16. $m.p_i^{old} := \Phi_{w+1}(m.p_i^{old})$;
17. $p_i^{old} := \emptyset$;

Step 14) initializes the processing of the next window ($w+1$). Step 15) removes facts that have been selected for processing (in Step 14) from $m.p_i^{new}$. Steps 16) - 17) discard facts from window $w$ that have been processed and are no longer needed. We illustrate this evaluation algorithm with an example.

Example 4.1 Consider again the Fibonacci program of Example 2.1 with the query $fib(n, X)$. The result of applying Magic Templates to this program appears in Example 2.1. We now turn to evaluating this program using Sliding Window Tabulation.

First we consider the down phase. Recall that the windowing function here is $\phi(m.fib(N)) = N$, the window size is 2, and that if the original query was $fib(n, X)$, the bounds are $c_0 = n - 2$ and $c_1 = 0$. Table 1 gives the values for the relations in question at the end of each iteration in the "down" phase for the query $fib(5, X)$.

The starting window for the "down" phase is $W_2$. Table 2 gives the values for the relations in question at the end of each iteration in the "up" phase.

4.4 Properties

First, we verify that Sliding Window Tabulation correctly evaluates programs that have the sliding window property.

Theorem 4.1 Let $(P, q)$ have the sliding window property. Then Sliding Window Tabulation computes all answers to $q$ and terminates.

Next, we turn to the efficiency of the tabulation. One key advantage of seminaive as compared to naive is that it never repeats a derivation. This property is known as the "seminaive property." In the next theorem we show that this is true of Sliding Window Tabulation also has the seminaive property.

Theorem 4.2 Let $(P, q)$ have the sliding window property. Then Sliding Window Tabulation has the seminaive property.
The overall goal of Sliding Window Tabulation is to limit the storage required by the program evaluation. The following set of definitions, culminating in Theorem 4.3, give bounds on the space efficiency of Sliding Window Tabulation.

**Definition 4.1** Let \( (P, q) \) have the sliding window property, with \( \phi \) the ordering function on the facts and goals of \( P^{m9} \). Also, let \( P \) be the set of goals and facts produced in the Sliding Window Tabulation of \( P^{m9} \). Then the goal width of \( (P, q) \) is the maximum, over all constants \( c \), of the number of goals \( g \in P \) such that \( d(g) = c \). Similarly, the fact width of \( (P, q) \) is the maximum, over all constants \( c \), of the number of facts \( f \in P \) such that \( d(f) = c \). The width of \( (P, q) \) is the larger of the goal or fact widths for \( (P, q) \).

**Definition 4.2** Let \( (P, q) \) have the sliding window property, with \( \phi \) the ordering function on the goals and facts of \( P \). Then the goal span of \( (P, q) \) is the maximal value \( s \) such that for some rule firing in the Sliding Window Tabulation of \( P^{m9} \), goal \( m.p_1 \) appears in the head, \( p_2 \) appears in the body, and \( \phi(m.p_1) - \phi(p_2) = s \). Similarly, the fact span of \( (P, q) \) is the maximal value \( s \) such that for some rule firing in the Sliding Window Tabulation of \( P^{m9} \), fact \( p_1 \) appears in the head, \( p_2 \) in the appears in the body, and \( \phi(p_1) - \phi(p_2) = s \). The span of \( (P, q) \) is the larger of the goal or fact spans of \( (P, q) \).

**Definition 4.3** Let \( (P, q) \) have the sliding window property. Then the basis width \( b \) of \( (P, q) \) is the number of relevant basis facts determined by the down phase of the sliding window evaluation of \( (P, q) \).

**Theorem 4.3** Suppose that \( (P, q) \) has width \( w \), span \( s \), height \( h \), and basis width \( b \). Then Sliding Window Tabulation stores at most \( w(s + h) + b \) goals or facts at any given time.

**Corollary 4.1** Suppose \( (P, q) \) has constant width, span, height, and basis width, and furthermore that any goal or fact of \( (P, q) \) can be stored in constant space. Then Sliding Window Tabulation runs in constant space.

---

<table>
<thead>
<tr>
<th>Window</th>
<th>Iteration</th>
<th>( b.m.fib^{old} )</th>
<th>( m.fib^{old} )</th>
<th>( m.fib^{save} )</th>
<th>( m.fib^{range} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>init</td>
<td>{5}</td>
<td>{5}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>{(4),(3)}</td>
<td>{(5)}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>{(5),(4),(3)}</td>
<td>{(2),(1)}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>init</td>
<td>{(2)}</td>
<td>{(4),(3),(2)}</td>
<td>{(1)}</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>{(2)}</td>
<td>{(4),(3),(2)}</td>
<td>{(1),(0)}</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>init</td>
<td>{(3)}</td>
<td>{(5),(3),(2)}</td>
<td>{(0)}</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>{(3)}</td>
<td>{(5),(3),(2)}</td>
<td>{(0)}</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>init</td>
<td>{(0)}</td>
<td>{(5),(3),(2)}</td>
<td>{(0)}</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>{(2),(1)}</td>
<td>{(5),(3),(2)}</td>
<td>{(0)}</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>0</td>
<td>0</td>
<td>{(0),(1)}</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: "Down" phase of evaluation of \( fib(5, X) \).

Example 4.2 Returning to the Fibonacci example, we have that \( s = 2 \), \( w = 1 \), \( h = 2 \), and \( b = 2 \). Since each fact is an integer and each goal is a pair of integers, if we assume that an integer can be stored in constant space, then Sliding Window Tabulation is constant space on Fibonacci. An example of a program on which Sliding Window Tabulation runs in linear space is given in Section 5.

The time efficiency of Sliding Window Tabulation is more difficult to analyze than the space efficiency. The simplest way to calibrate the performance of Sliding Window Tabulation on \( (P, q) \) appears to be a comparison with the seminaive evaluation of \( P^{m9} \). Even this comparison is not straightforward, for the following reasons:

1. In some cases, sliding window "overtabulates". That is, it may compute facts that are not computed by seminaive evaluation of \( P^{m9} \); these facts are not relevant to the query.

To understand why, consider the "up" phase. This phase is initialized using the set of relevant basis facts. Subsequently, the modified original rules in \( P \) and the inverted magic rules are fired repeatedly. This eliminates the need to store all magic facts from the "down" phase, but it raises the possibility of computing irrelevant program facts. Intuitively, this happens when some magic fact \( m \) can be generated from two distinct magic facts, say \( m_1 \) and \( m_2 \), where only one of the magic facts was produced on the way down. On the way back up, there is no way to know which of the two magic facts produced \( m \) on the way down, so both are generated on the way up.

2. There are overheads in Sliding Window Tabulation that are not present in seminaive evaluation of \( P^{m9} \). For example, when a new fact \( f \) is produced, we must evaluate \( \phi(f) \) before deciding where the fact should be saved. As another example, when "sliding" the window, any facts in the saved relations that belong in the new current window must be found, and moved from the save relation to the corresponding "new" relation.

3. On the other hand, often the number of facts stored by sliding window tabulation is much less than that stored...
Table 2: “Up” phase of evaluation of $fib(5, X)$.

<table>
<thead>
<tr>
<th>Window</th>
<th>Iteration</th>
<th>$\delta fib^{old}$</th>
<th>$fib^{old}$</th>
<th>$\delta m.fib^{old}$</th>
<th>$m.fib^{old}$</th>
<th>$m.fib^{save}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>init</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${(0),(1)}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>${(0,1),(1,1)}$</td>
<td>$\emptyset$</td>
<td>${(2)}$</td>
<td>${(0),(1)}$</td>
<td>${(3)}$</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>${(2,2)}$</td>
<td>${(0,1),(1,1)}$</td>
<td>$\emptyset$</td>
<td>${(0),(1),(2)}$</td>
<td>${(3),(4)}$</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>$\emptyset$</td>
<td>${(0,1),(1,1),(2,2)}$</td>
<td>$\emptyset$</td>
<td>${(0),(1),(2)}$</td>
<td>${(3),(4)}$</td>
</tr>
<tr>
<td>1</td>
<td>init</td>
<td>$\emptyset$</td>
<td>${(1,1),(2,2)}$</td>
<td>${(3)}$</td>
<td>${(1),(2)}$</td>
<td>${(4)}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>${(3,3)}$</td>
<td>${(1,1),(2,2)}$</td>
<td>$\emptyset$</td>
<td>${(1),(2),(3)}$</td>
<td>${(4),(5)}$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$\emptyset$</td>
<td>${(1,1),(2,2),(3,3)}$</td>
<td>$\emptyset$</td>
<td>${(1),(2),(3)}$</td>
<td>${(4),(5)}$</td>
</tr>
<tr>
<td>2</td>
<td>init</td>
<td>$\emptyset$</td>
<td>${(2,2),(3,3)}$</td>
<td>${(4)}$</td>
<td>${(2),(3)}$</td>
<td>${(5)}$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>${(4,5)}$</td>
<td>${(2,2),(3,3)}$</td>
<td>$\emptyset$</td>
<td>${(2),(3),(4)}$</td>
<td>${(5),(6)}$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$\emptyset$</td>
<td>${(2,2),(3,3),(4,5)}$</td>
<td>$\emptyset$</td>
<td>${(2),(3),(4)}$</td>
<td>${(5),(6)}$</td>
</tr>
<tr>
<td>3</td>
<td>init</td>
<td>$\emptyset$</td>
<td>${(3,3),(4,5)}$</td>
<td>${(5)}$</td>
<td>${(3),(4),(5)}$</td>
<td>${(6),(7)}$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>${(5,8)}$</td>
<td>${(3,3),(4,5)}$</td>
<td>$\emptyset$</td>
<td>${(3),(4),(5)}$</td>
<td>${(6),(7)}$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>$\emptyset$</td>
<td>${(3,3),(4,5),(5,8)}$</td>
<td>$\emptyset$</td>
<td>${(3),(4),(5)}$</td>
<td>${(6),(7)}$</td>
</tr>
</tbody>
</table>

We can, however, prove the following two theorems. The first, Theorem 4.4, gives a worst case upperbound on the number of inferences; the second, Theorem 4.5, shows that much better performance can be guaranteed if the magic rules in $P^{m9}$ are "invertible".

**Theorem 4.4** Sliding Window Tabulation of $(P, q)$ never infers more program facts than the seminaive evaluation of $P$.

Note that the above theorem only addresses the set of program facts that are inferred; it can be extended by noting that Sliding Window never infers more facts than seminaive evaluation of $P$ plus the magic facts inferred in seminaive evaluation of $P^{m9}$.

**Definition 4.4** [Invertibility] A magic rule $r$ in $P^{m9}$ is said to be invertible if the following holds. Suppose that $r$ is instantiated so that the head is $m1$ and the (only) magic fact in the body is $m2$. Given $m1$ and $r$, we should be able to determine $m2$.

**Theorem 4.5** Suppose that all the magic rules in $P^{m9}$ are invertible, and that the Seminaive evaluation of $P^{m9}$ makes $M$ magic fact inferences and $P$ program fact inferences. Then Sliding Window Tabulation of $(P, q)$ makes at most $2M$ magic fact inferences and $P$ program fact inferences.

### 5 An Example

Deductive database technology has been proposed as a useful tool in DNA sequence analysis. The Longest Common Subsequence (LCS) is representative of some of the low-level problems that are involved in this type of analysis. We are given two strings, say $A = a_0 a_1 \ldots a_{m-1}$, and $B = b_0 b_1 \ldots b_{n-1}$, where the $a_i$ and $b_j$ are drawn from some common alphabet. The desired answer is the maximal $x$ such that there is a string $C = c_0 c_1 \ldots c_{x-1}$, where $C$ is a subsequence of both $A$ and $B$. Note that "subsequence" differs from "substring" in that the members of a subsequence $C$ of $A$ and $B$ need not appear contiguously in either $A$ or $B$; all that is required is that the elements of $C$ appear in the same order in $A$ and $B$.

Hirchberg [Hir75] gives the following program to compute the LCS of two strings. (This is also discussed by Bird [Bir80] in the context of tabulation.) To express the problem in logic programming notation, we represent the string $A = a_0 a_1 \ldots a_{m-1}$ by the facts $a(0, a_0), a(1, a_1), \ldots, a(m-1, a_{m-1})$. Similarly, the string $B = b_0 b_1 \ldots b_{n-1}$ is represented as $b(0, b_0), b(1, b_1), \ldots, b(n-1, b_{n-1})$. 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 $X$.

\[
lcs(m, N, 0).
\]

\[
lcs(M, n, 0).
\]

\[
lcs(M, N, X) :- M < m, N < n, a(M, C), b(N, C),
\]

\[
lcs(M + 1, N + 1, X - 1).
\]

\[
lcs(M, N, X) :- M < m, N < n, a(M, C), b(N, D), C > D,
\]

\[
lcs(M + 1, N, X1), lcs(M, N + 1, X2),
\]

\[
X = \max(X1, X2).
\]

The longest common subsequence of the two strings is given by the query $lcs(0, 0, X)$.

First, if we use Prolog to evaluate this query, in the worst case the running time is $O(mn^2)$. As noted in the introduction, this is impractically large for all but the smallest $m$ and $n$. Another approach to evaluating the query is to use Magic Templates to rewrite the program, then to evaluate the result bottom-up. The resulting Magic rules are:

\[
m.lcs(1,1).
\]

\[
m.lcs(M + 1, N + 1) :- M < m, N < n, a(M, C), b(N, C),
\]

\[
m.lcs(M, N).
\]

\[
m.lcs(M + 1, N) :- M < m, N < n, a(M, C), b(N, D),
\]
query algorithm will choose there are two bound arguments in the magic program, there
algorithm 3.1 attempts to find a windowing function 4. Since to verify that the LCS program does indeed have the sliding
window property by running Algorithm 3.1. First, Algorithm
are six choices for 4. The Sliding Window Detection al-
complexity remaining O(mn)).

In order to apply Sliding Window Tabulation, we first need to verify that the LCS program does indeed have the sliding
window property by running Algorithm 3.1. First, Algorithm
there are six choices for 4. The Sliding Window Detection al-
program predicates on any derived program predicates. If
the magic predicates of one clique may depend upon
magic predicates on any derived program predicates. If
to extend Sliding Window Tabulation to deal with such
procedures.

The Independence condition for the applicability of Slid-
ing Window Tabulation disallows the dependence of
magic predicates on any derived program predicates. If
a program P contains more than one recursive clique, the
magic predicates of one clique may depend upon
program predicates from other cliques. It is desirable
to extend Sliding Window Tabulation to deal with such

6 Conclusion

We have presented a broad framework for compile-time
integration with general bottom-up evaluation. For example, it is likely that while tabulation
may be possible to design suitable window only at run-time. A good example is a program
that traverses an acyclic graph, say a part-subpart hier-
archy, and (possibly) does some additional computation.
In such cases, an interesting problem is to devise dy-
namic strategies that, possibly through some auxiliary
run-time computation and/or additional stored facts, identify a set of windows that result in significant space
savings overall.

• Refining sliding window techniques.
The techniques presented here can be refined in many
ways, including devising stronger tests for applicabil-
ity, and developing techniques for "sliding" windows in
greater increments, thereby minimizing the processing of
windows that contain few or no facts.

• Multiple recursive cliques.
The Independence condition for the applicability of Slid-
ing Window Tabulation disallows the dependence of
magic predicates on any derived program predicates. If
a program P contains more than one recursive clique, the
magic predicates of one clique may depend upon
program predicates from other cliques. It is desirable
to extend Sliding Window Tabulation to deal with such

• Dynamic methods.
Sliding Window Tabulation is a static method in that it
tries to determine window functions 4 at compile time.
Sometimes, it may be possible to design suitable win-
dows only at run-time. A good example is a program
that traverses an acyclic graph, say a part-subpart hier-
archy, and (possibly) does some additional computation.
In such cases, an interesting problem is to devise dy-
namic strategies that, possibly through some auxiliary
run-time computation and/or additional stored facts, identify a set of windows that result in significant space
savings overall.

• Integrating with general bottom-up evaluation.
Finally, a number of issues must be addressed in order to
incorporate the tabulation techniques investigated here
into a system based upon rewriting and seminaive eval-
uation. For example, it is likely that while tabulation
is not applicable to the entire program, it is applica-
table to a subprogram. To deal effectively with this sit-
uation, techniques must be developed to integrate the
optimizations that are possible for the subprogram into
the evaluation of the entire program.
The tradeoff between recomputation and storage has received little attention in the domain of deductive database programs, and to our knowledge has not been addressed at all in the context of bottom-up evaluation strategies. This paper demonstrates the potential gains from considering this problem by presenting bottom-up evaluation schemes that avoid recomputation without saving every intermediate result for the duration of the computation.

**Acknowledgements**

Divesh Srivastava and S. Sudarshan made several valuable comments on an earlier draft of this paper.

**References**


