Indexed Step-Wise Semi-Naive Evaluation for Recursive Queries

Ke Wang
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore, 0511
Email: wangk@iscs.nus.sg
Weining Zhang
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada, T1K 3M4
zhang@cs.uleth.ca

Abstract

A nice property of the semi-naive evaluation of recursive rules is that it does not repeat rule firings in different iterations. In a recent study [W], however, it was observed that intermediate steps during a rule firing may be repeated in different iterations, and a notion of the step-wise semi-naive property and an evaluation with this property were proposed to capture the semi-naive property at intermediate steps of a rule firing. In this paper, we further extend the semi-naive notion to the build-up of index for computing relations that grow monotonically in each iteration. Hash tables are chosen as the index in this study, but the idea can be applied to other types of index in general. A modification of the step-wise semi-naive evaluation based on the semi-naive build-up of hash tables is proposed. In most cases, join, set difference, and set union operations performed to fire a rule are implemented in a single I/O scan of relations that are cumulated up to that iteration. Experiments conducted on a few typical recursive queries and test data show that the proposed evaluation is much more I/O cost effective than both the semi-naive evaluation and the step-wise semi-naive evaluation.

1 Introduction

An iterative bottom-up evaluation of recursive rules computes the least fixpoint by firing rules in iterations until no new facts can be derived. The term "rule firing" here refers to instantiation of all subgoals with established facts so that all constraints in the rule body are satisfied. An evaluation has the *semi-naive* property if no rule firings in different iterations are duplicated. The basic semi-naive evaluation was rediscovered by several researchers [B, BalR1, Bayer] and was generalized and improved a few times [BalR2, KNSS, RSS]. Essential to these variations of the semi-naive evaluation are rewriting each rule into a number of differential or semi-naive versions and firing them through use of relational join operation. Recently, the following observation regarding semi-naive evaluations was drawn [W]: although the semi-naive property guarantees that no rule firing as a whole will be duplicated in subsequent iterations, it does not guarantee that no intermediate steps of a rule firing will be so duplicated. Consider the join-based implementation $(a \bowtie b) \bowtie \Delta c$ of a differential rule $p :- a, b, \Delta c$ in a semi-naive evaluation, whereby predicates a, b, c are recursive subgoals, Δc is the incremental version of c (arguments are omitted for simplicity). In iteration i, the intermediate relation $a_i \bowtie b_i$ was computed and was discarded immediately after it is joined with Δc_i. When computing intermediate relation $a_{i+1} \bowtie b_{i+1}$ in iteration $i+1$, all facts in $a_i \bowtie b_i$ are recomputed because definite rules are monotonic (thus $a_i \subseteq a_{i+1}$ and $b_i \subseteq b_{i+1}$). In other words, though the semi-naive evaluation performs incremental computation for relations defined by rules, it fails to do the same for relations generated at intermediate steps of rule firings, such as the result of $a \bowtie b$ in the above example. To address the semi-naive notion at intermediate steps of an evaluation, the *step-wise semi-naive* property was defined and a step-wise semi-naive evaluation was presented in [W]. The essence of a step-wise semi-naive evaluation is that join of every two tuples during a rule firing is not duplicated in different iterations (see Section 2 for more details). This is achieved by storing intermediate relations computed at internal nodes of the evaluation tree for a recursive rule. It has been analytically shown [W] that much fewer tuples are generated in the step-wise semi-naive evaluation than in the semi-naive evaluation, a measure of efficiency recommended by [BR].

However, the step-wise semi-naive evaluation still suffers from inefficiency because implementation issues at access path level are not considered. Firstly, the evaluation gives no instruction on how indexes of relations can be shared across iterations to speed up the query processing. For performing a large join efficiently, it is desirable to make use of certain indexes, such as hash tables in hash-based join [Bra] or sorted lists in merge/sort join [BE]. Since current relations in any two consecutive iterations differ only by an increment computed in one iteration, the join index in each iteration

can be obtained from the join index used in the last iteration by reflecting only the increment. Secondly, at each intermediate step the step-wise semi-naive evaluation needs to perform one set difference and one set union to remove "new" tuples that were previously generated, in order to enforce the step-wise semi-naive property. The cost of these operations is ignored in the analysis of [W] where only the number of generated tuples, or equivalently, the number of successful rule firings, is taken into account. Since there are no immediate implementations of these set operations that use a single scan of operand relations [1] (note that the result of set union is required to be free of duplicates), performing these set operations at each intermediate step of a rule firing in a naive way could incur a heavy I/O cost in the case where databases can not be held in the memory. Finally, measuring only the number of tuples generated can be largely inaccurate when the work of unsuccessful rule firings is substantial.

In this paper, a new bottom-up evaluation of recursive rules is proposed to remedy the above problems. The proposed evaluation aims at minimizing the I/O cost while preserving the step-wise semi-naive property by building up indexes of growing relations in a semi-naive manner, thus the name of *indexed step-wise semi-naive evaluation*. In this study, we choose hash tables as the index, but the methodology is applicable to other indexes as well. By assuming that each joining pair of buckets of increments can be held in the memory, which is reasonable in most cases, the I/O cost of each iteration is only a single I/O scan of the current relations partially computed up to that iteration, where an I/O scan of a relation refers to either reading or writing all pages of the relation once. This is the best one can possibly expect in the absence of preprocessing and knowledge of data distribution. In contrast, five such scans are required for the step-wise semi-naive evaluation and a number of scans depending on the number of recursive subgoals in the rule is required for the semi-naive evaluation. Experiments conducted on some typical recursive queries and test data also show that the proposed evaluation is much more I/O cost effective than both the semi-naive evaluation and the step-wise semi-naive evaluation. As its major shortcoming, however, the proposed evaluation requires more storage space, as compared to the other two evaluations. In Section 6, we will remark on this issue and consider possibilities of reducing the storage requirement by sharing hash tables among nodes of evaluation trees.

The rest of the paper is organized as follows: In Section 2 we briefly examine the ideas and merits of the semi-naive evaluation and the step-wise semi-naive evaluations through examples. In Section 3, we identify I/O inefficiency of the step-wise semi-naive evaluation and propose the indexed step-wise semi-naive evaluation aiming at reducing I/O cost. In Section 4, the I/O cost of the indexed step-wise semi-naive

[1] In many systems, set difference and duplicate elimination of set union are implemented through sorting.

evaluation is analyzed and is analytically compared with the step-wise semi-naive evaluation and the basic semi-naive evaluation. A comparison based on experiments is given in Section 5. In Section 6 we remark on space requirement of the proposed evaluation, and in Section 7 we conclude the paper.

2 Existing Bottom-up Evaluations

Before proposing a new evaluation of recursive rules, we review briefly some existing bottom-up evaluations of recursive rules. We shall restrict to *stratified rules* [BR, U], so that there is a partial order among blocks of mutually recursive predicates by which relations of recursive predicates can be monotonically evaluated over iterations. This shall include all rules evaluable by the naive or semi-naive bottom-up evaluations [BR]. It is assumed that the reader is familiar with the terminology in the area of deductive databases, such as rules, EDB and IDB predicates, subgoals, least fixpoint, etc. For more background information of deductive databases, please refer to [BR, U].

We shall stress that optimization strategies that rewrite rules to make their evaluations more efficient do not fall into the class of evaluations; only actual query evaluation algorithms will be considered here. A quick look at the list provided by [BR] tells that only the naive evaluation and the semi-naive evaluation are purely bottom-up and of general application domains. In this paper, we are interested in the semi-naive evaluation and the step-wise semi-naive evaluation recently proposed in [W].

2.1 Basic Semi-Naive Evaluation

A few variations of the semi-naive evaluation have been proposed recently, based on improved strategies of evaluating loops [KNSS, RSS]. But their essence remains the same as the basic semi-naive evaluation, as illustrated by the following example of a simple recursive query.

Example 2.1 Assume that a recursive query has four rules:

α : $t(x,y) :- t(x,w), t(w,u), s(w,u), e_1(u,y)$
β : $s(x,y) :- t(x,w), s(w,u), e_2(u,y)$
γ : $t(x,y) :- e_3(x,y)$
δ : $s(x,y) :- e_4(x,y)$

t and s are recursive predicates, and α and β are recursive rules. These rules contain only *ordinary* subgoals; subgoals that are not ordinary are defined on predicates $=, \neq, <, >, \leq, \geq$, called *built-in* subgoals. The *body* of a rule refers to the right side of $:-$ and the *head* of a rule refers to the left side of $:-$ in that rule. Figure 1 shows

the basic semi-naive evaluation of these rules. For each IDB predicate p, the *current relation* p holds the tuples computed in all previous iterations and the *increment* $\triangle p$ holds the tuples computed only in the last iteration. The evaluation terminates when all increments $\triangle p$ computed by some iteration are empty. Importantly, the evaluation avoids duplicating the same rule firing in different iterations by mapping rule α to three differential versions and mapping rule β to two differential versions, where each differential version replaces exactly one recursive subgoal with its increment computed in the last iteration. □

However, in the semi-naive evaluation intermediate computation of a rule firing could still be repeated in different iterations. For example, if the join order for the first differential version of rule α is $\Delta t \bowtie ((t \bowtie s) \bowtie e_1)$, computing $t_i \bowtie s_i$ in iteration i will duplicate the work of computing $t_{i-1} \bowtie s_{i-1}$ that was previously done in iteration $i-1$. One may argue that this problem will disappear if these joins are executed in a linear order that starts with the increment, i.e., $((\Delta t \bowtie t) \bowtie s) \bowtie e_1$ for the above joins. But such increment-headed linear join orders are usually contradictory with other optimization requirements. For instance, if the evaluated rule is a magic rule in which the underlying side-way information passing strategy [BMSU] requires evaluating the joins in a rule body from left to right, then it is impossible to meet both ordering requirements (i.e., the increment-headed linear order and the side-way information passing order) in evaluating the third differential version $\prod_{x,y}(t \bowtie t \bowtie \Delta s \bowtie e_1)$ of rule α. Also, the choice of linear join ordering completely rules out the possibility of parallelizing multiway joins in a rule as in [Gr]. A desirable evaluation should allow join evaluation plans that are more flexible than linear joins for the sake of further optimizations. These observations have motivated the proposal of the step-wise semi-naive property in [W].

2.2 Step-Wise Semi-Naive Evaluation

The step-wise semi-naive evaluation was proposed in [W] as an improvement of the basic semi-naive evaluation, by ensuring that no intermediate steps of rule firings are duplicated in different iterations. Consider the rules in Example 2.1 and assume the join order $t \bowtie ((t \bowtie s) \bowtie e_1)$ for evaluating rule α and the join order $(t \bowtie s) \bowtie e_2$ for evaluating rule β, as given by the *evaluation trees* [W] in Figure 2. Before the loop begins, relations for $t, s, \Delta t, \Delta s$ are initialized by

$t \leftarrow e_3$;
$\Delta t \leftarrow t$;
$s \leftarrow e_4$;
$\Delta s \leftarrow s$

begin
$t \longleftarrow e_3$;
$\triangle t \longleftarrow t$;
$s \longleftarrow e_4$;
$\triangle s \longleftarrow s$;
repeat
$\triangle t' \longleftarrow \prod_{x,y}(\triangle t \bowtie t \bowtie s \bowtie e_1)$
$\cup \prod_{x,y}(t \bowtie \triangle t \bowtie s \bowtie e_1)$
$\cup \prod_{x,y}(t \bowtie t \bowtie \triangle s \bowtie e_1)$;
$\triangle s' \longleftarrow \prod_{x,y}(\triangle t \bowtie s \bowtie e_2)$
$\cup \prod_{x,y}(t \bowtie \triangle s \bowtie e_2)$;
$\triangle t \longleftarrow \triangle t' - t$;
$t \longleftarrow t \cup \triangle t$;
$\triangle s \longleftarrow \triangle s' - s$;
$s \longleftarrow s \cup \triangle s$;
until $\triangle t = \emptyset$ and $\triangle s = \emptyset$;
output t and s
end.

Figure 1: Basic semi-naive evaluation of rules in Example 2.1

as before. Intermediate relations u_1, u_2, v_1 associated with non-root internal nodes of the two evaluation trees are explicitly stored throughout the evaluation. The least fixpoint is computed in relations t, s by the loop:

repeat
$X_t \leftarrow \emptyset$;
$X_s \leftarrow \emptyset$;
$X_t \leftarrow X_t \cup INCR(\alpha)$;
$X_s \leftarrow X_s \cup INCR(\beta)$;
$\Delta t \leftarrow X_t - t$;
$t \leftarrow t \cup \Delta t$;
$\Delta s \leftarrow X_s - s$;
$s \leftarrow s \cup \Delta s$;
until $\Delta t = \emptyset$ and $\Delta s = \emptyset$.

Functions $INCR(\alpha)$ and $INCR(\beta)$, defined below, are invoked to compute new tuples defined by α and β in each iteration, respectively.

$INCR(\alpha)$:
begin

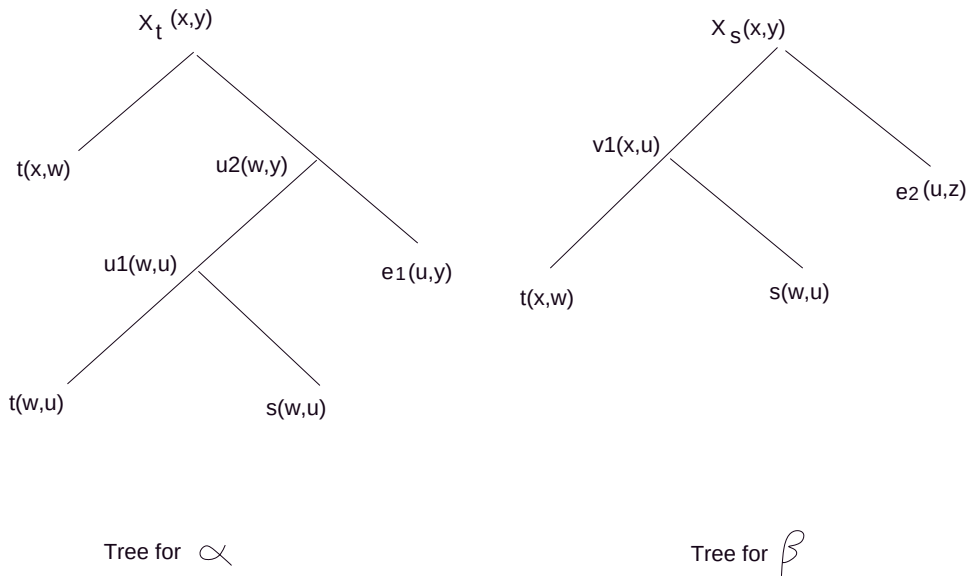


Figure 2: Evaluation trees for rules α and β

$\triangle u_1 \longleftarrow \prod_{w,u}(\triangle t(w,u) \bowtie s(w,u)) \cup \prod_{w,u}(t(w,u) \bowtie \triangle s(w,u))$;
$\triangle u_1 \longleftarrow \triangle u_1 - u_1$;
$u_1 \longleftarrow u_1 \cup \triangle u_1$;
$\triangle u_2 \longleftarrow \prod_{w,y}(\triangle u_1(w,u) \bowtie e_1(u,y))$;
$\triangle u_2 \longleftarrow \triangle u_2 - u_2$;
$u_2 \longleftarrow u_2 \cup \triangle u_2$;
return the result of $\prod_{x,y}(\triangle t(x,w) \bowtie u_2(w,y)) \cup \prod_{x,y}(t(x,w) \bowtie \triangle u_2(w,y))$;
end.

$INCR(\beta)$:
begin
$\triangle v_1 \longleftarrow \prod_{x,u}(\triangle t(x,w) \bowtie s(w,u)) \cup \prod_{x,u}(t(x,w) \bowtie \triangle s(w,u))$;
$\triangle v_1 \longleftarrow \triangle v_1 - v_1$;
$v_1 \longleftarrow v_1 \cup \triangle v_1$;
return the result of $\prod_{x,y}(\triangle v_1(x,u) \bowtie e_2(u,y))$;
end.

Intuitively, at the beginning of iteration i ($i \geq 1$), relations t and s contain all tuples derived in all iterations j for $j \leq i-1$, and increments Δt and Δs contain tuples that are derived in iteration $i-1$ but not derived in any iteration j for $j < i-1$.

During iteration i, each join operation derives new tuples using at least one new tuple, by having an increment as one of its operands. This amounts to propagating increments from leafs to the root of the evaluation tree, or equivalently from subgoals towards the head of the rule. Since each join involves one operand that contains only new tuples, no intermediate steps of firing rules are duplicated in different iterations. This is made possible by storing intermediate relations (such as u_i, v_i) produced by each join to avoid recomputation in subsequent iterations. In [W] this property of being free of duplication at every intermediate step of rule firings was called the *step-wise semi-naive* property and an evaluation with this property, called the step-wise semi-naive evaluation, was presented.

The above step-wise semi-naive evaluation assumes that an evaluation tree is constructed for each recursive rule with at least two ordinary subgoals. In general, each leaf of an evaluation tree represents a subgoal of the rule and each internal node represents a θ-join of the two children followed by a projection. More precisely, if an internal node $w(w_1, \ldots, w_l)$ has two children $u(u_1, \ldots, u_m)$ and $v(v_1, \ldots, v_n)$ with a set θ of inequalities on arguments of w (equalities are specified through shared variables as explained below), then node v defines the operations

$$\prod_{w_1,\ldots,w_l} (\Delta u(u_1, \ldots, u_m) \bowtie v(v_1, \ldots, v_n)) \cup \prod_{w_1,\ldots,w_l} (u(u_1, \ldots, u_m) \bowtie \Delta v(v_1, \ldots, v_n))$$

where

- $\prod_{w_1,\ldots,w_l}$ is projection,
- $\bowtie$ is θ-join,
- u, v are current relations, and
- $\Delta u, \Delta v$ are increments.

If θ is empty, $\bowtie$ becomes the natural join with the join attributes specified by variables common to both operands. To simplify a little, notation $u \bigoplus v$ will be used to denote the above operations associated with node w. Given the topology of the evaluation tree for a recursive rule, which is usually determined by certain optimization strategies such as side-way information passing or optimal order of joins, a construction of evaluation trees was presented in [W] with certain optimalities guaranteed. It is required in that construction that rules are safe, normalized, and connected, as defined below, but these are achievable by transformation from safe rules.

Definition 2.1 ([U]) A rule is *safe* if (a) every variable found in the head is found in the body, and (b) for every variable X not found in ordinary subgoals, there is some variable Y to which X is equated through a sequence of one or more $=$ subgoals, and

Y is either equal to some constant in a $=$ subgoal or is an argument of an ordinary subgoal. □

Definition 2.2 ([W]) A safe rule is *normalized* if it has no $=$ subgoal.□

Definition 2.3 ([RBK]) Two predicate instances are *connected* if they share a variable. The set of predicate instances in a rule, including the head, is partitioned into connected components. A rule is *connected* if it has only one connected component. □

Safety is necessary for ensuring that the finiteness of EDB relations implies the finiteness of the least fixpoint. It was shown in [W] that every safe rule can be efficiently transformed into an equivalent, safe, and normalized rule by reversing the rectification process [U], i.e., equating arguments whose equalities are logical consequence of the $=$ subgoals in the rule. A set of rules in which not every rule is connected can be replaced with a set of connected rules whose fixpoint evaluation is more efficient [RBK].

From now on, we consider only rules that are safe, normalized, and connected. We shall assume that, for each recursive rule with at least two ordinary subgoals, an evaluation tree is constructed as in [W] without further explaining the detail of the construction. For modification and comparison in subsequent sections, we reproduce the step-wise semi-naive evaluation of recursive rules from [W] in Figure 3.

3 Indexed Step-Wise Semi-Naive Evaluation

The straightforward implementation of the step-wise semi-naive evaluation as above has some major inefficiency. At each internal node v of an evaluation tree, the following operations are performed in every iteration

$$\Delta v \leftarrow R \bigoplus S;$$
$$\Delta v \leftarrow \Delta v - v;$$
$$v \leftarrow v \cup \Delta v;$$

where R and S are the child relations of v and Δ versions are increments. Besides the lack of information on use of indexes as mentioned in the introduction, this computation requires to perform one set difference $-$ and one set union $\cup$ at *each* internal node of an evaluation tree. Set difference and set union are usually more expensive than projection and selection because they have no immediate implementations that use only a single scan of operand relations (recall that the result of set union must be free of duplicates). In a large environment where databases can not be held in the memory, this will translate into a heavy I/O cost and degrade drastically the performance of the evaluation. Implementations that are substantially more efficient

than simply embedding invocation of relational operations in a loop of conventional programming languages are crucial to the development of deductive databases. *In this section, we propose an evaluation that performs all of the above join, projection, set difference, and set union associated with an internal node v in a single I/O scan of the current relations for the children of v. Three ideas make this possible: (a) have a (small) increment as an operand in every join performed, as offered by the step-wise semi-naive evaluation, (b) preserve the index of current relations for use in the next iteration, and (c) delay performing $\Delta v \leftarrow \Delta v - v$ and $v \leftarrow v \cup \Delta v$ until v is scanned as an operand relation of a later join in the tree.* (a) helps avoid iterative scans of the target relation in performing join. (b) restricts the work of building indexes to only increments. (c) combines the scans for join and set operations associated with an internal node into a single scan. These will be further elaborated below.

In this paper, we choose hash tables as the index, but the idea is not limited to hash tables. Unlike the conventional use of performing only join, however, hash tables here are used for performing join, set difference, and set union associated with internal nodes. Each internal node will be associated with a hash function to hash-partition its operand relations (i.e., child relations). If no hash-partitioning is chosen for an internal node, the hash function is assumed to map every tuple to the same bucket. The choice of hash functions can be different for different internal nodes.

Consider the computation in iteration i ($i \geq 1$). Let v be any internal node of the evaluation tree, and R, S be the two children of v. v, R, S also denote the current relations associated with nodes v, R, S. Let $\Delta v, \Delta R, \Delta S$ be the increments associated with nodes v, R and S. In the case of a leaf node R (similarly for S), ΔR holds the increment computed in iteration $i-1$ for the subgoal represented by leaf R, and relations R and ΔR contain only tuples that satisfy all equalities and inequalities on arguments of that subgoal, by first applying selections on these relations. In the case of an internal node R (similarly for S), ΔR holds the increment computed in iteration i for node R. Let h be the number of buckets defined by the hash function chosen for (the operation of) internal node v. Then $R^1, \ldots, R^h$ denote the buckets of R, and $\Delta R^1, \ldots, \Delta R^h$ denote the buckets of ΔR. Similarly, $S^1, \ldots, S^h, \Delta S^1, \ldots, \Delta S^h$ for S. h is usually determined by the size of available memory (because one page of memory is allocated to each bucket as output buffer during the hash-partitioning) and the type of join associated with node v. If the join predicate involves inequalities $<, >, \neq, \leq, \geq$, hash-partitioning should not be used for the join and the number h must be chosen as 1.

Small Increment Assumption: We shall assume that at each node v the increment Δv, computed in the latest iteration, is "much smaller" than the current relation v, cumulated in all previous iterations. This assumption is valid when the number of iterations needed in a bottom-up evaluation is large, which is usually the

case in deductive databases. When it comes to implementation, this has the following implications: for every internal node having children R and S,

- the memory can hold each pair of joining buckets ΔR^i and ΔS^i of increments ΔR and ΔS, and still enough memory is left for input and output buffers of the join operation. A nice feature of the existing step-wise semi-naive evaluation is that, for every join performed, one of the two operands is an increment. Then with each bucket of the increment held entirely in the memory, it is possible to perform a join by scanning the current relation only once. For details, see the indexed step-wise semi-naive evaluation below and its analysis in Section 4. However, we do not require the memory to hold buckets R^i or S^i of current relations R and S at once.

- within each iteration, the number of I/O scans of current relations R and S should be minimized, even at the cost of a few more scans of increments ΔR and ΔS.

To make life easier, for a recursive rule r, we shall use the term "a node (resp., a leaf, the root) of r" as the abbreviation of "a node (resp., a leaf, the root) of the evaluation tree of r". Now we are ready to present a new evaluation of recursive rules.

Indexed Step-Wise Semi-Naive Evaluation: The algorithm is shown in Figures 4 and 5. In the main program on top of Figure 4, the current relation p for each recursive predicate p is initialized by all non-recursive rules defining p (line 1), and a hash table is created for each leaf of an evaluation tree (line 2). In creating a hash table for a leaf representing a p-subgoal, the hash function associated with its parent is applied to tuples of relation p, which are then relocated into buckets according to computed hash values. It is possible that hash tables of two or more leafs (in the same or different trees) are shared to save storage space. For simplicity at the moment, we assume that a distinct hash table is created for each leaf; sharing of hash tables will be commented in Section 6. Hash tables of current relations of all internal nodes are initialized to empty (line 3).

Within each iteration, the algorithm computes the increment for each head predicate one at a time and updates hash tables associated with internal nodes of evaluation trees. In particular, for each recursive rule r considered, the algorithm calls $INCR(r)$ to propagate increments from leafs to the root of r (line 6). The increment computed at the root of r is cumulated by $INCR(r)$ in a special relation ΔX_p, where p is the head predicate of r. The main loop on top of Figure 4 terminates when condition $empty^r = true$ holds for all recursive rules r at the end of some iteration (line 7), indicating that at the beginning of the last iteration the increment of each leaf is a subset of the current relation of that leaf, and therefore, that no new tuples were

generated in the last iteration. The value of $empty^r$ is set by $INCR(r)$. When the main loop terminates, the algorithm outputs all recursive relations p in form of hash tables.

$INCR(r)$: (a) build hash tables for increments of recursive subgoals of r, (b) propagate increments from leafs to the root of r, (c) update hash tables of current relations to reflect their increments. For better efficiency, *unit rules*, i.e, rules with a single ordinary subgoal, and *non-unit rules*, i.e., rules with more than one ordinary subgoal, are treated differently. For a unit rule r, $UnitRule(r)$ essentially computes some projections and selections of the increment of the leaf. For a non-unit rule r, increments are computed bottom-up from leafs to the root in a partial order induced by the evaluation tree of r, where the processing at an internal node v is done by $Node(r, v)$.

$Node(r, v)$: propagate increments from children R and S to the parent v. If v is the root of r, the increment for v is stored in the special relation ΔX_p where p is the head predicate of r, otherwise, it is stored in Δv. This different treatment of the root allows increments for the same head predicate p produced by all recursive rules for p to be stored in the same relation, i.e, ΔX_p, so as to perform hash-partitioning and duplicate elimination only once. To produce the increment at v, the algorithm reads pairs of buckets ΔR^i and ΔS^i from disk, $1 \leq i \leq h$, one pair at a time. By the small increment assumption, $\Delta R^i, \Delta S^i$ can be held in the memory at once. For each pair $\Delta R^i, \Delta S^i$ read, buckets R^i and S^i are scanned page by page. Each scanned page of R^i is joined with ΔS^i with the result appropriately projected, denoted $\prod(RP^i \bowtie \Delta S^i)$. Similarly, each scanned page S^i is joined with ΔR^i with result appropriately projected, denoted $\prod(\Delta R^i \bowtie SP^i)$. These results are appended to the increment Δv (which was initialized to empty at the beginning of $Node(r, v)$) or ΔX_p on disk, depending on whether v is the root of r. During the same scan, $\Delta R \leftarrow \Delta R - R$ is computed by removing from ΔR^i tuples that are in R^i (line 4), and $R \leftarrow R \cup \Delta R$ is computed by appending the updated ΔR^i to R_i (line 5); similar operations are performed for S^i and ΔS^i (lines 8,9). $\prod(\Delta R^i \bowtie \Delta S^i)$ is also computed and appended to Δv or ΔX_p for each pair $\Delta R^i, \Delta S^i$ read (line 10).

To reduce I/O cost, when the result of join is appended to a file on disk, each disk writing is triggered by one of two conditions: either the output buffer is full or the data in the output buffer is the last page of the result. After all pairs of buckets are scanned, $empty^r_R$ (resp., $empty^r_S$) is set to true if R (resp., S) is a recursive subgoal and the increment ΔR (resp., ΔS) is empty (lines 11 and 12). This can be easily tested by checking if every updated ΔR^i (resp., ΔS^i) is empty at line 4 (resp., line 8). Then the disk storage for hash tables of ΔR and ΔS is released because increments generated in the next iteration will share no tuples with old increments. Finally, if v is not the root of r, the increment Δv is hash-partitioned and duplicates are removed

bucket by bucket (line 14). Since each bucket of an increment can be held entirely in the memory, duplicate elimination within a bucket of Δv can be done in memory. For the increment of the root, the hash-partitioning and duplicate elimination is done after all recursive rules defining the same predicate are evaluated, that is, in the next iteration where the increment is scanned as that of a leaf.

It is worth mentioning that ΔS participates in join (line 3) before duplicates are removed from it (line 8). For ΔR, however, duplicates are removed before joined. This implies that the indexed step-wise semi-naive evaluation approximates the step-wise semi-naive property rather than strictly enforces it. As a result, duplication of work may exist at intermediate steps of rule firings in different iterations. Similar approximation arises in the approximated semi-naive evaluation where no set difference is performed to remove old tuples [BR] (for the reason that set difference in general is not cheap.) In our case, duplicate tuples *are* eliminated (after used) in each iteration, and if there are some duplicates participating in the join, they must be generated only by a single iteration. We expect that the approximation in the proposed evaluation be better than the case of approximated semi-naive evaluation where duplicates are never removed. In most cases, duplication of work introduced by a single iteration is smaller than the overhead of additional scans of the whole current relations, and hence this sacrifice of the strict step-wise semi-naive property is worthwhile in order to achieve a better performance in term of the I/O cost.

Example 3.1 Consider the rules in Example 2.1 and the evaluation trees in Figure 2. Assume that domains of all arguments are integers. Hash functions h_1, h_2, h_3 associated with operations at internal nodes $u_1, u_2, X_t(x, y)$ can be chosen as

$h_1(w, u) = (w + u) \bmod 10$
$h_2(u) = u^2 \bmod 10$
$h_3(w) = w^2 \bmod 10,$

and hash functions h'_1, h'_2 associated with operations at internal nodes $v_1, X_s(x, y)$ can be chosen as

$h'_1(w) = w^3 \bmod 10$
$h'_2(u) = u \bmod 10.$

For example, with $i = (a + b) \bmod 10$, hash function h_1 will map each tuple $t(a, b)$ for subgoal $t(w, u)$ and each tuple $s(a, b)$ for subgoal $s(w, u)$ to the ith bucket of their hash tables. Figures 6 and 7 give the indexed step-wise semi-naive evaluation of these rules where only $Node(\alpha, u_1)$ and $Node(\alpha, X_t(x, y))$ are expanded; the other calls can be expanded similarly. □

4 Experimental Comparison

Experiments have been conducted on UNIX machines using the language C to supplement the analytical performance study of the three evaluations in Section 4. In all experiments, disk storage was simulated by the UNIX file system (relations are represented by UNIX files) and a memory of 1Mbytes is allocated for each evaluation. Each I/O page has 4 Kbytes and is viewed as an array of tuples, each tuple being 128 bytes. Therefore, the memory has 2^8 I/O pages of space. For each experiment we must (1) choose a set of recursive rules that will represent our benchmark query, (2) choose some test data that will represent the EDB relations, (3) collect the information about the I/O cost incurred in each evaluation of the recursive rules. Three experiments have been conducted. The first experiment evaluates non-linear recursive rules, the second evaluates linear recursive rules, and the third evaluates a non-linear version of transitive closure. The choice of EDB relations in each experiment will be explained later. Each time a page is read from or written to a disk file, it is counted as one page I/O.

We assume that domains of all arguments are integers and that the following hash function is used throughout to hash-partition relations:

$$f(w) \quad = \quad w \bmod h$$

where w refers to the value of a single join attribute or if there are more than one join attributes, the sum of their values. h, the number of partitioned buckets, is chosen to be 2^{m-1} if there are 2^m pages of memory available. In our case, $h = 128$.

We shall use the abbreviations BSN, SWSN, ISWSN for the basic semi-naive evaluation, the step-wise semi-naive evaluation, and the indexed step-wise semi-naive evaluation. In all graphs plotted in this section, coordinate (x, y), for positive integers x, y, denotes the cumulative I/O cost y in number of pages at the end of xth iteration. In the indexed step-wise semi-naive evaluation, there is an extra I/O cost incurred to construct the least fixpoint from hash tables of relations after the loop terminates. This extra I/O cost is included in the cost of the last iteration.

4.1 Experiment 1

The first experiment is designed to measure the I/O cost incurred by various evaluations on the non-linear rules in Example 2.1. For convenience, we reproduce those rules below

$$\begin{aligned} \alpha : t(x,y) \quad &:- \quad t(x,w),\ t(w,u),\ s(w,u),\ e_1(u,y). \\ \beta : s(x,y) \quad &:- \quad t(x,w),\ s(w,u),\ e_2(u,y). \end{aligned}$$

$$\begin{aligned} \gamma : t(x,y) &:- e_3(x,y). \\ \delta : s(x,y) &:- e_4(x,y). \end{aligned}$$

Evaluation trees in Figure 2 are used for rules α and β. Two runs of the experiment were conducted, each handling a different set of test data. In both runs, EDB relations $e_1, \ldots, e_4$ are identical and are binary trees, that is, a tree in which each node has at most two children. To model branch variation of binary trees, we use

$$\textit{fan-out ratio } a : b$$

to denote that, at an internal node v, the probability that v has one child is a and the probability that v has two children is b. For example, the extreme case of a linear chain is modeled by ratio $1 : 0$ and the extreme case of complete binary tree is modeled by ratio $0 : 1$. The first run of the experiment was conducted on the linear chain of depth 250, and the second run on the binary tree of fan-out ratio $0.8 : 0.2$ with depth of 35. The tree used in the second run was generated top-down starting with the root, and at each node a decision is make regarding whether one or two children are produced according to the fan-out ratio $0.8 : 0.2$. There are a total of 574 tuples generated for each EDB tree in the second run. The performance of BSN, SWSN, and ISWSN in both runs are illustrated by Figures 8 and 9, respectively.

First of all, we observe that ISWSN always needs one more iteration than BSN and SWSN because ISWSN delays its set difference and set union operations on recursive predicates till the next iteration. Despite this, the result of this experiment shows that ISWSN performs significantly better than BSN and SWSN. In fact, as the number of iterations increases, the improvement in performance increases too. Two closely related factors contribute to the saving in I/O cost in ISWSN: the semi-naive build-up of hash tables and the semi-naive computation of intermediate relations throughout the evaluation. These two factors together allow the join, set difference and set union operations to be performed in a single I/O scan of current relations at each internal node of an evaluation tree.

4.2 Experiment 2

The second experiment investigates the performance of various evaluations on linear rules. We borrow the the following linear rules and test data from in [KRS]:

$$\begin{aligned} t(x,y,z) &:- e_1(x,y,z). \\ t(x,y,z) &:- e_2(x,u), t(u,y,z). \\ t(x,y,z) &:- e_2(y,v), t(x,v,z). \\ t(x,y,z) &:- e_2(z,w), t(x,y,w). \end{aligned}$$

where

$$\begin{aligned} e_1 &= \{(x,y,z)|x = 100i, y = 100j, z = 100k; i,j,k = 0,\ldots,m\} \\ e_2 &= \{(x,y)|x = y+1, y = 100i+j; i = 0,\ldots,m; j = 0,\ldots,n\} \end{aligned}$$

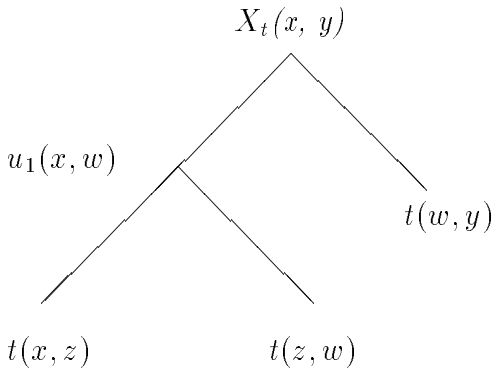
e_1 can be regarded as vectors in three dimensional space. They are arranged in a cubic grid, separated from each other by one hundred units, and contained in a cube that has opposite vertices of $(0,0,0)$ and $(100 \cdot m, 100 \cdot m, 100 \cdot m)$. The derived relation t consists of all vectors on the finest cubic grids extended from points in e_1, with the maximal extension in each direction being n. In this experiment, m and n are set to the value of 3 and 4 respectively. Figure 10 shows that in evaluating these linear rules, ISWSN performs significantly better than BSN and SWSN, because BSN and SWSN need additional I/O scans of current relations to hash-partition the current relations and perform set difference and set union operations. However, SWSN was observed to give the same performance as BSN. This is so because in the particular case of these linear rules, no intermediate relations, i.e., non-root internal nodes in the evaluation tree, are involved and SWSN degenerates to BSN.

4.3 Experiment 3

In the third experiment, we consider a non-linear version of the transitive closure. Transitive closure is often described as typical recursive queries and its non-linearization would enable to achieve increased degree of parallel computation, in the sense of building longer paths out of a few shorter paths. The following non-linear version of the transitive closure is considered:

$$\begin{aligned} t(x,y) &:- e(x,y). \\ t(x,y) &:- e(x,z), e(z,y). \\ t(x,y) &:- t(x,z), t(z,w), t(w,y) \end{aligned}$$

where the third rule derives the reachability of a path using reachability of three shorter paths. The readers can easily verify that these rules correctly compute the transitive closure of relation e. The evaluation tree for the recursive rule is as follows:



This experiment is conducted using the the binary tree with a fan-out ratio of 0.8:0.2 and a depth of 35. The result is shown in Figure 11, in which ISWSN performs better than BSN and SWSN until the extra iteration of ISWSN where the total cumulative I/O cost of ISWSN exceeds that of SWSN. A close examination reveals that this exceeding is caused by (a) duplication work due to the approximated step-wise semi-naive property of ISWSN and (b) the extra cost of combining buckets of hash tables to return the least fixpoint which we have included in the last iteration of ISWSN. However, as the number of iterations needed increases, the gain of reduced I/O scans through semi-naive build-up of hash tables will dominate the performance of ISWSN.

One may also notice that although Experiment 1 shows no big difference in I/O performance between SWSN and BSN, in Experiment 3 SWSN performs significantly better than BSN. The reason is that the evaluation trees in Experiment 1 have more internal nodes, i.e., 5, than the evaluation tree in Experiment 3, i.e., 2. The number of internal nodes reflects the number of set difference and set union performed in each iteration of SWSN. When there are more internal nodes the I/O saving by enforcing the step-wise semi-naive property in SWSN can be offset by the overhead of performing the additional set operations at internal nodes. However, the number of internal nodes does not affect the performance of ISWSN much because set operations are performed in the same I/O scan as for the join at each internal node.

In summary, the results of the three experiments have indicated that by semi-naive build-up of hash tables, we are able to reduce the I/O cost incurred in evaluating recursive rules. The results also indicate that three factors influence the performance of ISWSN: the type of queries or rules, the characteristics of EDB relations, and the amount of duplication work. In general, if the amount of duplication work introduced by a single iteration is not "very large" and if the number of iterations needed is "large", ISWSN is expected to perform significantly better than BSN and SWSN.

5 Conclusion

We now conclude the paper by highlighting the progress made by each of the semi-naive evaluation, the step-wise semi-naive evaluation, and the indexed step-wise semi-naive evaluation. Compared to the naive evaluation that repeatedly computes all previously computed facts plus new facts in each iteration, the semi-naive evaluation computes only new facts in each iteration. However, this primitive form of the semi-naive notion addresses only derivation of facts about recursive predicates, not intermediate tuples generated during a rule firing. The step-wise semi-naive evaluation extends the semi-naive notion to computation of all intermediate tuples so that each iteration computes only new intermediate tuples. The indexed step-wise semi-naive evaluation further extends the semi-naive notion to the build-up of indexes that are used at the data access level during the evaluation. This progressive enforcement of the semi-naive notion at the predicate level, the workspace level, and the data access level has leaded to a substantial performance improvement of least fixpoint evaluations.

The major drawback of the indexed step-wise semi-naive evaluation is its higher storage space requirement as compared to the semi-naive evaluation and the step-wise semi-naive evaluation. This is due to the fact that (a) intermediate relations produced at each step of rule firing is stored and (b) subgoals of the same predicate may not be able to share the same hash table. However, the strongest argument for the indexed step-wise semi-naive evaluation is that, except for storage limitation, it does not make sense to discard intermediate results or indexes that are known to be used at chance of 100% in the next iteration. As storage price continuously goes down, the constraint of storage requirement of the proposed evaluation is expected to become less and less visible.

References

[B] F. Bancilhon, "A note on the performance of rule based systems," MCC Technical Report DB-022-85

[BalR1] I. Balbin and K. Ramamohanarao, "A differential approach to query optimization in recursive deductive databases," TR-86/7, Dept. of Computer Science, University of Melbourne

[BalR2] I. Balbin and K. Ramamohanarao, "A generalization of the differential approach to recursive query evaluation," *J. Logic Programming*, Vol. 4, No. 2, 1987

[Bayer] R. Bayer, "Query evaluation and recursion in deductive database systems," unpublished manuscript, 1985

[BE] M.W. Blasgen and K.P. Eswaran, "Storage and access in relational databases," *IBM Systems Journal*, Vol. 16, No. 4, 1977

[BMSU] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman, "Magic sets and other strange ways to implement logic programs," in proceedings of *ACM Symposium on Principles of Database Systems*, Cambridge, MA, March 1986, pp. 1-15

[BR] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," in proceedings of the *ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 28-30, 1986, ACM, New York, pp. 16-50

[Bra] K. Bratbergsengen, "Hashing methods and relational algebra operations," in proceedings of *International Conference on Very Large Data Bases*, pp. 323-333, 1984

[FNPS] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, "Extendible hashing — A fast access method for dynamic files", *ACM transactions on Database Systems*, Vol. 4, No. 3, pp 315-344, sept. 1979

[Gr] G. Graefe, "Query evaluation techniques for large databases", *ACM Computing Surveys*, Vol. 25, No. 2, June 1993, pp. 73-170

[KNSS] J. Kuittinen, O. Nurmi, S. Sippu, and E. S. Soininen, "Efficient Implementation of loops in bottom-up evaluations of logic queries," in proceedings of *16th International Conference on Very Large Data Bases Conference*, Brisbane, Australia, pp. 372-379, 1990

[KRS] D.B. Kemp, K. Ramamohanarao, and Z. Somogyi, "Right-, left-, and multilinear rule transformations that maintain context information", in proceedings of the *16th International Conference on Very Large Data Bases*, Brisbane, Australia, pp 380-391, 1990

[L] P. Larson, "Dynamic hashing", *BIT*, 18 (1978)

[RBK] R. Ramakrishnan, C. Beeri, and R. Krishnamurthy, "Optimizing existential datalog queries," in proceedings of *ACM Symposium on Principles of Database Systems*, pp. 89-102, 1988

[RSS] R. Ramakrishnan, D. Srivastava, and S. Sudarshan, "Rule ordering in bottom-up fixpoint evaluation of logic programs," in proceedings of *16th International Conference on Very Large Data Base*, Brisbane, Australia, pp. 359-371, 1990

[S] Y. Sagiv, "Optimizing DATALOG programs," in *Foundations of deductive databases and logic programming* (J. Minker, Ed.), Morgan Kaufmann Publishers, Los Altos, CA, pp. 659-698, 1988

[U] J.D. Ullman, *Principles of database and knowledge base systems*, Vol. I and II, CSP, Rockville, Maryland, 1988

[W] K. Wang, "Step-wise semi-naive evaluation of recursive queries", Technical Report, Dept. of ISCS, National University of Singapore, 1994

Step-Wise Semi-Naive Evaluation: Evaluate a set of safe, connected, and normalized rules.
Precondition: An evaluation tree has been constructed for each recursive rule with at least two ordinary subgoals.
Method:
for each recursive predicate p **do** initialize both p and Δp by the exit rules for p; initialize current relations of all non-root internal nodes in all evaluation trees to empty set;
repeat
    for each recursive predicate p **do** $X_p \leftarrow \emptyset$;
    for each recursive rule r defining some recursive predicate p **do**
        $X_p \leftarrow X_p \cup INCR(r)$;
    for each recursive predicate p **do** $\{\Delta p \leftarrow X_p - p;\ p \leftarrow p \cup \Delta p\}$;
until $\Delta p = \emptyset$ for all recursive predicates p;
output relations p's for all recursive predicates p's.

function $INCR(r)$: a set of tuples for the head predicate of rule r
1. **if** r has a single ordinary subgoal, say of the form $q(\vec{Z}) :- p(\vec{X}), F$, where F is conjunction of the built-in subgoals
    then return the result of $\prod_{\vec{Z}}(\sigma_F(\Delta p(\vec{X})))$;
2. compute the current relations and incremental relations for all non-root internal nodes in the partial order induced by
    the evaluation tree for r. More precisely, the computation at node w denoting operation $u \bigoplus v$ is as follows:
    $\Delta w \leftarrow u \bigoplus v$;
    $\Delta w \leftarrow \Delta w - w$;
    $w \leftarrow w \cup \Delta w$;
3. **return** the result of $u \bigoplus v$, where $u \bigoplus v$ is the operation at the root.

Figure 3: Step-wise semi-naive evaluation

Indexed Step-Wise Semi-Naive Evaluation: Evaluate a set of mutually recursive rules that are safe, connected, and normalized.
Precondition: An evaluation tree has been constructed for each recursive rule with at least two different ordinary subgoals.

1. **for** each recursive predicate p **do** initialize p and ΔX_p by non-recursive rules for p;
2. create hash tables for current relations of all subgoals;
3. set empty all buckets of current relations of all non-root internal nodes;
4. **repeat**
5. **for** each recursive predicate p **do** $\{\Delta p \leftarrow \Delta X_p; \Delta X_p \leftarrow \emptyset\}$;
6. **for** each recursive predicate p **do for** each rule r defining p **do** $INCR(r)$;
7. **until** $empty^r = true$ for every recursive rule r;
8. output relations p, in form of hash tables on disk, for all recursive predicates p.

procedure $INCR(r)$
Input: r — a recursive rule with at least two ordinary subgoals;
Effect: cumulate increment for the head predicate p of r in X_p. Also update current relations associated with internal nodes of r.

1. **for** the increment Δp of each recursive subgoal in r **do**
   create a hash table and eliminate duplicates in each bucket of the table;
   if r is a unit rule **then** $UnitRule(r)$ **else**
   { **for** $i = 1$ to n **do** $Node(r, v_i)$, where $v_1, \ldots, v_n$ is a bottom-up partial order of all internal nodes of r;
   $empty^r \leftarrow empty^r_{s_1} \wedge \ldots \wedge empty^r_{s_k}$, where $s_1, \ldots, s_k$ are the recursive subgoals in r }.

procedure $UnitRule(r)$
Input: r — a unit recursive rule with the head predicate p and the recursive leaf R.
Effect: the result of $\prod(\sigma(\Delta R))$ is added to ΔX_p. Also, new tuples of ΔR are added to R.

for $i = 1$ to h **do** {
1. read ΔR^i;
   for each page of R^i, denoted RP^i, **do**
2. { read RP^i;
3. append the result of $\prod(\sigma(RP^i))$ to ΔX_p on disk, where selection σ corresponds to the built-in subgoals in r };
4. append ΔR^i to R^i on disk;
   };
5. **if** $\Delta R = \emptyset$ **then** $empty^r_R \leftarrow true$;
6. release the disk space of ΔR;

Figure 4: Indexed step-wise semi-naive evaluation

procedure $Node(r, v)$
Input:
r — a non-unit rule with head predicate p;
v — an internal node of r;
$R, \Delta R, S, \Delta S$ — child relations of node v, in form of hash tables stored on disk.
Effect: the result of $\prod(R \bowtie \Delta S) \cup \prod(\Delta R \bowtie S) \cup \prod(\Delta R \bowtie \Delta S)$ is stored in buckets of Δv if v is not the root of r, or is added to relation ΔX_p if v is the root of r. Also, new tuples of ΔR and ΔS are added into buckets of R and S on disk.

if v is not the root of r **then** $\Delta v \leftarrow \emptyset$;
for $i = 1$ to h **do**
{
1. read ΔR^i and ΔS^i;
for each page of R^i, denoted RP^i, **do**
{
2. read RP^i;
3. **if** v is the root **then** append the result of $\prod(RP^i \bowtie \Delta S^i)$ to ΔX_p on disk
else append the result of $\prod(RP^i \bowtie \Delta S^i)$ to Δv on disk;
4. remove in memory from ΔR^i all tuples that are in RP^i;
};
5. append ΔR^i to R^i on disk; /* $R \leftarrow R \cup \Delta R$ */
for each page of S^i, denoted SP^i, **do**
{
6. read SP^i;
7. **if** v is the root **then** append the result of $\prod(\Delta R^i \bowtie SP^i)$ to Δv on disk
else append the result of $\prod(\Delta R^i \bowtie SP^i)$ to Δv on disk;
8. remove in memory from ΔS^i all tuples that are in SP^i;
};
9. append ΔS^i to S^i on disk; /* $S \leftarrow S \cup \Delta S$ */
10. **if** v is the root **then** append the result of $\prod(\Delta R^i \bowtie \Delta S^i)$ to ΔX_p on disk
else append the result of $\prod(\Delta R^i \bowtie \Delta S^i)$ to Δv on disk;
};
11. **if** R is a recursive subgoal and $\Delta R = \emptyset$ **then** $empty_R^r \leftarrow true$;
12. **if** S is a recursive subgoal and $\Delta S = \emptyset$ **then** $empty_S^r \leftarrow true$;
13. release disk space of ΔR and ΔS;
14. **if** v is not the root **then** create the hash table for Δv and eliminate duplicates in each bucket of Δv.

Figure 5: Continued from Figure 4

$t \leftarrow e_3$;
$\Delta X_t \leftarrow t$;
$s \leftarrow e_4$;
$\Delta X_s \leftarrow s$;
create hash tables for current relations of leafs $t(x,w), t(w,u), s(w,u), e_1(u,y)$ of rule α by hash functions $h_3(w), h_1(w,u)$ $h_1(w,u), h_2(u)$, respectively;
create hash tables for current relations of leaf $t(x,w), s(w,u), e_2(u,z)$ of rule β by hash functions $h'_1(w), h'_1(w), h'_2(u)$, respectively;
initialize hash tables for current relations of nodes u_1, u_2, v_1 to empty;
repeat
    $\Delta t \leftarrow \Delta X_t$;
    $\Delta X_t \leftarrow \emptyset$;
    $\Delta s \leftarrow \Delta X_s$;
    $\Delta X_s \leftarrow \emptyset$;
    $INCR(\alpha)$;
    $INCR(\beta)$;
until $empty^\alpha = true \wedge empty^\beta = true$;
output relations t and s in form of hash tables on disk.

$INCR(\alpha)$
    create hash tables for leafs $\Delta t(x,w), \Delta t(w,u), \Delta s(w,u)$ by hash functions $h_3(w), h_1(w,u), h_1(w,u)$, respectively, and eliminate duplicates in each bucket;
    $Node(\alpha, u_1)$;
    $Node(\alpha, u_2)$;
    $Node(\alpha, X_t(x,y))$;
    $empty^\alpha \leftarrow empty^\alpha_{t(x,w)} \wedge empty^\alpha_{t(w,u)} \wedge empty^\alpha_{s(w,u)}$.

$INCR(\beta)$
    create hash tables for leafs $\Delta t(x,w), \Delta s(w,u)$ by hash functions $h'_1(w), h'_1(w)$, respectively, and eliminate duplicates in each bucket;
    $Node(\beta, v_1)$;
    $Node(\beta, X_s(x,y))$;
    $empty^\beta \leftarrow empty^\beta_{t(x,w)} \wedge empty^\beta_{s(w,u)}$.

Figure 6: Indexed step-wise semi-naive evaluation for Example 3.1

$Node(\alpha, u_1)$

$\Delta u_1 \leftarrow \emptyset$;
for $i = 1$ to 10 **do**
{ read $\Delta t(w,u)^i$ and $\Delta s(w,u)^i$;
for each page of $t(w,u)^i$, denoted RP^i, **do**
{ read RP^i;
append the result of $\prod_{w,u}(RP^i(w,u) \bowtie \Delta s(w,u)^i)$ to Δu_1^i on disk;
remove in memory from $\Delta t(w,u)^i$ all tuples that are in RP^i };
append $\Delta t(w,u)^i$ to $t(w,u)^i$ on disk;
for each page of $s(w,u)^i$, denoted SP^i, **do**
{ read SP^i;
append the result of $\prod_{w,u}(\Delta t(w,u)^i \bowtie SP^i(w,u))$ to Δu_2^i on disk;
remove in memory from $\Delta s(w,u)^i$ all tuples that are in SP^i };
append $\Delta s(w,u)^i$ to $s(w,u)^i$ on disk;
append the result of $\prod_{w,u}(\Delta t(w,u)^i \bowtie \Delta s(w,u)^i)$ to Δu_1^i on disk };
if $\Delta t(w,u) = \emptyset$ **then** $empty^{\alpha}_{t(w,u)} \leftarrow true$;
if $\Delta s(w,u) = \emptyset$ **then** $empty^{\alpha}_{s(w,u)} \leftarrow true$;
release disk space of $\Delta t(w,u)$ and $\Delta s(w,u)$;
create the hash table for Δu_1 by $h_2(u)$ and eliminate duplicates in each bucket.

$Node(\alpha, X_t(x,y))$

for $i = 1$ to 10 **do**
{ read $\Delta t(x,w)^i$ and Δu_2^i;
for each page of $t(x,w)^i$, denoted RP^i, **do**
{ read RP^i;
append the result of $\prod_{x,y}(RP^i(x,w) \bowtie \Delta u_2^i(w,y))$ to ΔX_t on disk;
remove in memory from $\Delta t(x,w)^i$ all tuples that are in RP^i };
append $\Delta t(x,w)^i$ to $t(x,w)^i$ on disk;
for each page of u_2^i, denoted SP^i, **do**
{ read SP^i;
append the result of $\prod_{x,y}(\Delta t(x,w)^i \bowtie SP^i(w,y))$ to ΔX_t on disk;
remove in memory from Δu_2^i all tuples that are in SP^i };
append Δu_2^i to u_2^i on disk;
append the result of $\prod_{x,y}(\Delta t(x,w)^i \bowtie \Delta u_2^i(w,y))$ to ΔX_t on disk };
bf if $\Delta t(x,w) = \emptyset$ **then** $empty^{\alpha}_{R} \leftarrow true$;
release disk space of $\Delta t(x,w)$ and Δu_2.

Figure 7: Continued from Figure 6

Algorithm	Total I/O cost in pages
ISWSN	221793
SWSN	312750
BSN	323302
size of least fixpoint in tuples: $t = 15750$, $s = 15750$	

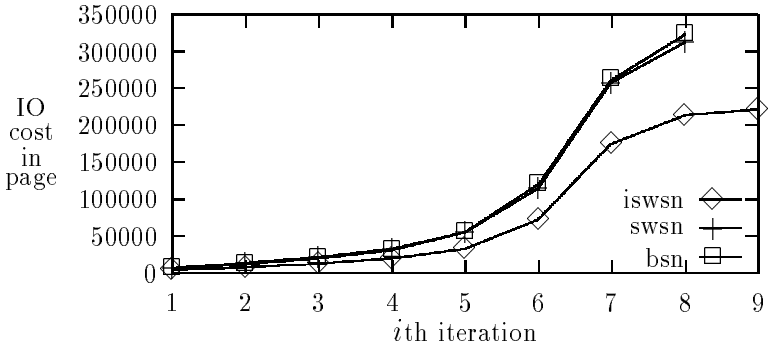


Figure 8: Result of Experiment 1 with linear chain of depth 250

Algorithm	Total I/O cost in pages
ISWSN	54887
SWSN	85504
BSN	89245
size of least fixpoint in tuples: $t = 8641$, $s = 8641$	

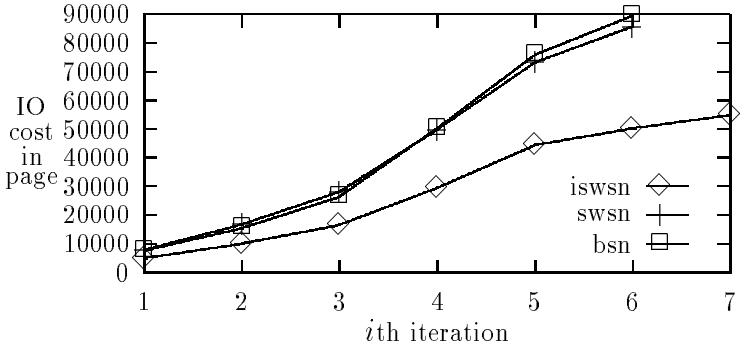


Figure 9: Result of Experiment 1 with fan-out ratio 0.8:0.2 and depth 35

Algorithm	Total I/O cost in pages
ISWSN	13443
SWSN	21347
BSN	21347
size of least fixpoint in tuples: 13824	

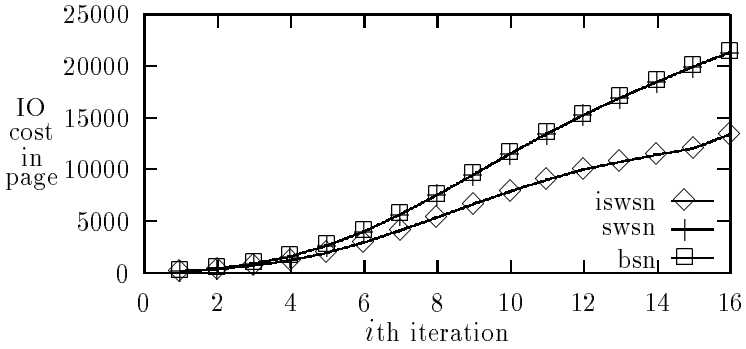


Figure 10: Result of Experiment 2 with m = 3 and n = 4

Algorithm	Total I/O cost in pages
ISWSN	210316
SWSN	147262
BSN	282616
size of least fixpoint in tuples: 17016	

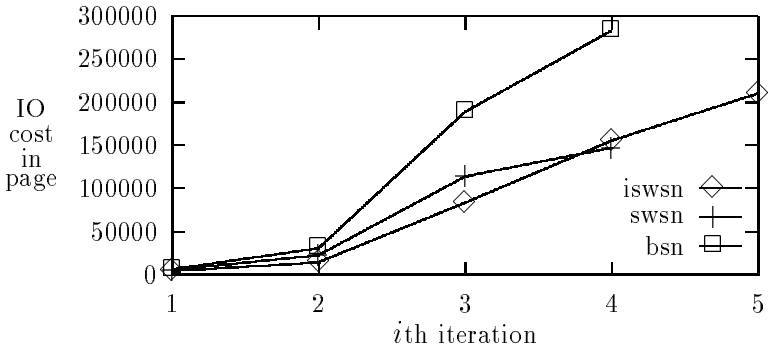


Figure 11: Results of Experiment 3 with fan-out ratio=0.8:0.2 and depth=35