ØN THE COMPUTATION OF THE TRANSITIVE CLOSURE OF RELATIONAL OPERATORS

by

Yannis E. Ioannidis

Memorandum No. UCB/ERL M86/51

25 May 1986

ON THE COMPUTATION OF THE TRANSITIVE CLOSURE OF RELATIONAL OPERATORS

by

Yannis E. Ioannidis

Memorandum No. UCB/ERL M86/51

25 May 1986

TITLE PAGE

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ON THE COMPUTATION OF THE TRANSITIVE CLOSURE OF RELATIONAL OPERATORS

by

Yannis E. Ioannidis

Memorandum No. UCB/ERL M86/51

25 May 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ON THE COMPUTATION OF THE TRANSITIVE CLOSURE OF RELATIONAL OPERATORS

Yannis E. Ioannidis

Department of Electrical Engineering and Computer Science
Computer Science Division
University of California
Berkeley, CA 94720

Abstract

Query processing in the presence of recursively defined views usually involves some form of iteration. For example, computing the transitive closure of a tree involves iterating N times, where N is the depth of the tree, each time computing pairs of vertices that are one edge further apart than the pairs produced in the previous iteration. Applying a divide and conquer technique we devise algorithms that need a logarithmic number of iterations. Assuming that we are looking for complete materializations of the recursively defined relations we show both through analytical and experimental results that this approach is in many cases superior in performance than the N-iteration algorithm.

This research was supported by the National Science Foundation under Grant ECS-8300463

1. INTRODUCTION

Recursion has received considerable attention in the context of deductive databases, i.e. databases in which new facts may be derived from ones that were explicitly introduced. Interesting work has been done on the formalization of the problem and how to perform query processing in the presence of recursion ([Hens84, Banc86, Han86, Rose86, Ullm85, Viei86] etc).

The purpose of this paper is to present some experimental results on recursion processing that indicate that the "obvious" algorithms usually used are not always optimal. In fact, as will be apparent in the sequel, we may use different algorithms and get significant improvements in both I/O and CPU time.

The paper is organized as follows. In Section 2 we give an overview of the operator model that we will use to describe recursion and its processing. Most of this section is an excerpt from [Ioan86] where the operator model was first introduced. Section 3 contains the description of a new algorithm together with an analysis of its I/O behavior as compared with that of the usual iterative algorithm. We also give some experimental performance results of the two algorithms. In Section 4, we elaborate on a whole class of algorithms that can be conceived expanding on our initial idea. Some limited performance results are also given there. Finally, Section 5 includes some of our current ideas on algorithms for more general classes of recursion, whereas Section 6 concludes with some possible directions for future research in the development of query optimizers for recursion.

We assume that the reader is familiar with 1-st order logic [Ende72] and Horn clauses [Gall84], which are usually the basis for the study of recursion in a database environment.

2. OPERATOR MODEL

In this paper we will be using the following canonical example for recursion. Consider a database with a relation **father** with schema **father**(fath,son). Using **father** in the following two Horn clauses we may define the virtual relation **ancestor** with schema **ancestor**(anc,desc):

$$\textbf{ancestor}(x,z) \wedge \textbf{father}(z,y) \rightarrow \textbf{ancestor}(x,y)$$

$$\textbf{father}(x,y) \rightarrow \textbf{ancestor}(x,y)$$

The first Horn clause is recursive in the sense that the relation **ancestor** appears on both the qualification and the consequent of it. Answering queries on relations defined recursively as **ancestor** above is the problem we want to address.

For the purpose of this paper we concentrate on *linear* and *immediate* recursion. This means that we have a single recursive Horn clause and the recursive predicate appears in the antecedent only once. Dissallowing any function symbols such a recursive Horn clause will have the form

$$P(\underline{x}^{(0)}) \wedge Q_1(\underline{x}^{(1)}) \wedge \cdots \wedge Q_k(\underline{x}^{(k)}) \rightarrow P(\underline{x}^{(k+1)}) \qquad (1)$$

where for each i, $\underline{x}^{(i)}$ is a subset of some fixed set of variables $\{x_1,x_2,...,x_n\}$ and P is a virtual relation and $\{Q_i\}$ a fixed set of stored relations. As analyzed in [Ioan86], the problem of recursion can be defined in operator form as follows. A recursive Horn clause, like the one shown above, may be considered as some relational operator A applied on some relation P to produce more tuples for the same relation. So, it can be written more clearly as $AP \subseteq P$, where A is an operator mapping relations over a fixed set of domains into ones over the same set of domains. The relations in $\{Q_i\}$ will be parameters of the operator A. If we employ this approach we are able to define operations on relational operators as follows. *Multiplication* of operators is defined by

$$(A * B)P = A(BP)$$

and *addition* by

$$(A+B)P = AP \cup BP$$

For notational convenience we omit the operator $*$. Identity ($1P = P$) and null ($0P = \emptyset$, $\emptyset$ the empty set) are defined in obvious ways. The n-th power of an operator A is inductively defined as:

$$A^0 = 1, \; A^n = A*A^{n-1} = A^{n-1}*A$$

Having established an algebraic framework, the problem of immediate recursion can now be stated as follows: Assume that we have a recursive Horn clause that can be represented by the

operator A, so that

$$AP \subseteq P$$

Also, there exists some constant relation R, which is either stored or produced by some other set of Horn clauses not involving P, so that

$$R \subseteq P$$

Then, the relation defined by the given set of Horn clauses can be found as the solution to the equation

$$P = AP \cup R \tag{2}$$

Presumably, the solution will be a function of R; we can write $P = BR$ and the problem becomes one of finding the operator B. Manipulation of (2) will result in the elimination of R so that we have an equation of operators only. In this pure operator form the recursion problem can be restated as follows: Given some operator A, find another one B satisfying:

(a) $1 + AB = B$

(b) B is minimal with respect to (a), i.e. for all other C satisfying (a), it is $B \leq C$.

The solution to the above equation (a) under the constraint (b) was shown in [Ioan86] to be equal to

$$A^* = \sum_{k=0}^{\infty} A^k.$$

The operator A^* is called the *transitive closure* of A, which taking into account the definitions of $+$ and $*$ may also be written as

$$A^* = \lim_{k \to \infty} (1 + A)^k. \tag{3}$$

Since A does not contain any functions, for every finite relation R there exists some n_0 (depending on R) such that

$$A^*R = \sum_{k=0}^{n_0} AR = (1 + A)^{n_0} R.$$

This says the fortunate and somewhat obvious fact that when dealing with finite relations (which is the case in a database environment) only a finite number of the terms of the sum are enough to

give us the complete result. Hence, A^* is an operator mapping finite relations to finite relations.

3. NEW ALGORITHMS

As we showed in the previous section a relation defined recursively can be materialized by applying some operator A^* on some stored relation R. However, there are many equivalent forms for A^*, each one suggesting a different algorithm for computing A^*R. Some of these equivalent forms together with the corresponding algorithms are studied in the next subsection.

3.1. Equivalent Forms of the Solution

In section 2 we have already mentioned two equivalent forms for A^*, namely $\lim_{k\to\infty}(1+A)^k$ and $\sum_{k=0}^{\infty}A^k$. These correspond exactly to the two main algorithms suggested in the literature for processing recursion.

In retrospect, examining the original algorithm that has been proposed to solve recursion [Aho79], we can see that it is a direct application of

$$A^* = \lim_{k\to\infty}(1+A)^k = \cdots(1+A)(1+A)$$

(whenever explicit parenthesization is omitted, right associativity is assumed for multiplication). In other words, we apply A on some initial relation R and then we take the union of the result with R and apply A on that etc. until nothing new gets produced.

As an example consider A being the operator corresponding to the Horn clause defining the **ancestor** relation of section 2. Assume that the relation **father** is given by the graph of figure 3.1, where an edge $(a \rightarrow b)$ indicates a tuple **father**(a,b), i.e. a is the father of b. Consider the query **ancestor**(Uranus,y), i.e. we are looking for the descendents of Uranus. Following the steps of the algorithm corresponding to (3) we see the answer being developed as shown in figure 3.1.

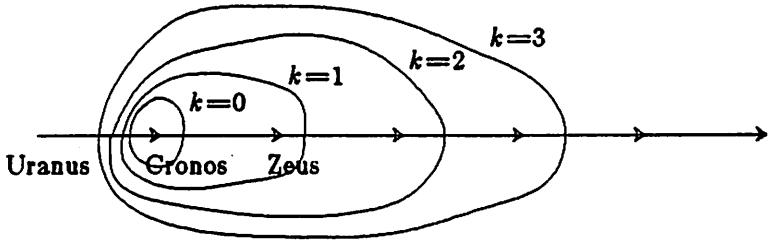

Figure 3.1. Naive algorithm to answer **ancestor**(Uranus,y).

The answer is developed by way of finding in the k-th step ($(1 + A)^k$) Uranus's descendents that are l generations below him, $1 \leq l \leq k+1$, for $k \geq 0$. In other words initially we get Uranus's children, then in step 1 we get them with Uranus together to find their children, which are Uranus's children and grandchildren together etc. This is clearly not a very good way of doing things since A is applied on the same tuples many times producing the same result again and again. For example Cronos is produced as a descendent of Uranus in every iteration.

In that respect the form $A^* = \sum_{k=0}^{\infty} A^k$ is easily seen as more practical in the sense of being faster. Exactly this form has been followed in [Gutt84] and elsewhere. Also, this form has been formally extracted in [Baye84] and [Banc85], even for more general cases of recursion, i.e. without any assumptions about linearity. To understand the algorithm implied by that we have to see that we can write the above equation as

$$A^* = \sum_{k=0}^{\infty} A^k = 1 + (1 + (1 + (\ \cdots\ )A)A)A \tag{4}$$

In this form A is applied each time only on the tuples produced during the previous iteration, so unnecessary computation is avoided.

For the same example as before, using A^* in its form of (4) we generate Uranus' descendents as shown in figure 3.2.

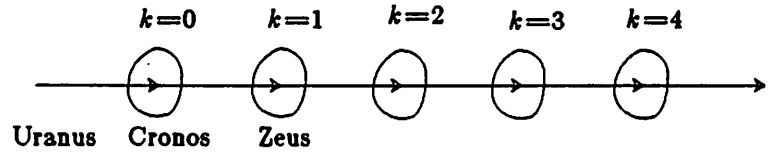

Figure 3.2. Semi-naive algorithm to answer **ancestor**(Uranus,y).

In the k-th step, A^k, $k \geq 0$, we find Uranus' descendents that are exactly $k+1$ generations below him, i.e. initially we find Uranus' children, in step 1 we find their children, which are Uranus' grandchildren etc. In the end we take the union of all the sets to get the complete answer.

In our search for other equivalent forms for A^* that will possibly give us more efficient execution algorithms we arrive at the following form:

$$A^* = \prod_{k=0}^{\infty}(1 + A^{2^k}) = \cdots (1 + A^4)(1 + A^2)(1 + A) \qquad (5)$$

The algorithm indicated by this formula for A^* avoids the application of the same operator on the same tuples more than once so it is presumably faster than the original formula (3). The interesting question is how it compares with the second formula (4). To get a feeling for this algorithm we use it to find again Uranus' descendents. The steps of the generation of the answer are shown in figure 3.3.

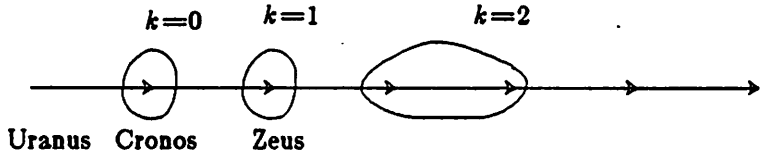

Figure 3.3. Smart algorithm to answer **ancestor**(Uranus,y).

Using (5), in the k-th step we generate Uranus' descendents that are l generations below him, $2^{k-1} < l \leq 2^k$, $k \geq 0$, i.e. we get Uranus' children, then in step 1 we take them with Uranus and get their grandchildren, which are Uranus' grandchildren and great-grandchildren together etc. The particular query that we used in our examples is not one where (5) is the most efficient as it will be pointed out in section 6. It is only presented here because of its simplicity.

At this point we should mention that the algorithm corresponding to (5) was independently proposed by Valduriez and Boral [Vald86]. They use a different formalism to describe it, namely Relational Algebra Programs, but it is essentially the same algorithm as (5). We should also mention that this algorithm is reminiscent of graph theory algorithms to find transitive closure of graphs.

Adopting the terminology of [Banc85] we call the algorithm corresponding to (3) *naive algorithm* and the one to (4) *semi−naive algorithm*. In the same spirit we call the algorithm corresponding to (5) *smart algorithm*. Looking at the three algorithms together we see the following: At each step the naive algorithm applies the same operator on all the tuples that have been produced during the execution. The semi-naive algorithm does the same but only on the tuples produced in the last iteration. Finally the smart algorithm at each step applies a different operator on all the tuples produced up to that point. In this sense it is the dual of the semi-naive algorithm with respect to the naive algorithm. Figure 3.4 summarizes the above.

	Same Operator	Different Operator
Last iteration result	*SEMI-NAIVE*	
Complete result	*NAIVE*	*SMART*

Figure 3.4. Algorithm types for the computation of A^*.

It is an interesting question to study whether there exists an algorithm covering the last remaining empty box.

The issue raised is whether this new smart algorithm will run any faster than the others. Looking at the formulas as given above is not enough to give us any useful conclusion in this direction. On the one hand, the number of multiplications performed by the smart algorithm is much smaller than by the semi-naive one (roughly it should be equal to $2*\log_2 N$, $\log_2 N$ to find the powers of 2 and another $\log_2 N$ for the outer multiplications, with N the number the semi-naive algorithm needs). Assuming that in most cases multiplying implies joining, we see that the smart algorithm performs fewer joins. On the other hand, in each step we are calculating bigger portions of the final outcome, by applying a more expensive operator on larger relations than the semi-naive algorithm. Hence, each step is definitely more expensive.

When this trade-off of the number of multiplications versus their individual costs is beneficial for the overall performance, is the focal point of the discussion that follows.

3.2. I/O Cost Analysis

In this section we present an analysis of the I/O performance of the semi-naive and smart algorithms mentioned above (formulas (4) and (5)). For simplicity, our analysis (and our experiments in section 3.3) have been restricted on A representing the computation of the transitive closure of a binary relation. However, our analysis extends easily to more general forms of A. We have assumed that for the implementation of the semi-naive algorithm we first compute all significant powers of A as $A^k = A\,A^{k-1}$ and at the end we take the sum of all of them. The relation in A is sorted only once in the beginning. Likewise for the smart algorithm we compute all operators A^{2^k} keeping the results sorted (**loop1**), and then perform massive joins of the current result with the corresponding power of A, which would be of the form A^{2^k}, for some k (**loop2**). The outcome of these joins is appended directly to the current result to be used in the next iteration.

For the analysis we need to define the following parameters.

$tup[k]$	Number of tuples in the relation of A^k
$page[k]$	Number of pages occupied by $tup[k]$ tuples (= $pagenum(tup[k])$, see below)
$pagenum(t)$	Number of pages occupied by t tuples
$pgsz$	Number of tuples fitting in a page
buf	Number of buffers available in the system
$sort(p)$	$= 2p\lceil \log_{buf} p \rceil$, I/O cost to sort p pages having buf buffers [Blas76]
C	I/O cost to create a relation
D	I/O cost to destroy a relation
N	Number of iterations needed by the semi-naive algorithm
M	$= \lfloor \log_2 N \rfloor = \max_k \{k: tup[2^k] \neq 0\}$

Notice that according to the above definitions, the final outcome for A^* should contain $\sum_{k=1}^{N} tup[k]$ tuples. Using the above parameters the I/O cost *s_naive_io* and *smart_io* of the semi-naive and smart algorithms respectively should be as follows:

semi-naive algorithm

$s\_naive\_io =$

$sort(page[1])$	Sort original relation on appropriate field(s). It is done only once.
$+ N\,page[1]$	At each step read sorted original relation.
$+ \sum_{k=1}^{N} sort(page[k])$	Sort the second relation for the join.
$+ \sum_{k=2}^{N} page[k]$	Write the outcome of the join.
$+ \sum_{k=1}^{N} page[k] + pagenum(\sum_{k=1}^{N} tup[k])$	At the end read all the intermediate results and put them into one relation.
$+ N(C + D) + C$	Create and Destroy the N intermediate results and also create the final result.

smart algorithm

$smart\_io =$

	loop1
$2\sum_{k=0}^{M} sort(page[2^k])$	For each step sort A^{2^k} on two different (set of) fields for the join. We keep one of them for loop2.
$+ \sum_{k=1}^{M} page[2^k]$	Write the result.
	loop2
$+ \sum_{k=1}^{M} sort(pagenum(\sum_{l=1}^{2^k-1} tup[l]))$	At each step sort current answer for the next join.
$+ \sum_{k=1}^{M} page[2^k]$	Read second relation (A^{2^k}), which is sorted from loop1.
$+ \sum_{k=1}^{M} pagenum(\sum_{l=1}^{2^k-1} tup[2^k+l])$	Join the two relations and append the outcome to the result.
$+ \sum_{k=1}^{M} page[2^k]$	Append the relation of the used A^{2^k} to the result.
$+ M(C + D) + C$	Create and destroy the M intermediate results and also create the final result.

Before evaluating the above formulas for some specific values of their parameters we would like to make the following comments.

(a) First, we can see that for large relations the most significant terms for both formulas should be the ones of sorting, since the other ones are linear in the size of the input. In the smart

algorithm we are sorting bigger relations than in the semi-naive one. Hence, we should expect that as the relations grow bigger the semi-naive algorithm should have better performance.

(b) On the other hand, since the sorting cost is highly dependent on the number of available buffers, increasing them should make the smart algorithm benefit more than the semi-naive one and therefore make its performance more competent.

(c) Also, for small relations (where sorting is not that expensive) the overhead of creating and destroying temporaries (costs C and D above) may become significant. In that case, the fact that the factor of $C + D$ in *s_naive_io* is greater than in *smart_io* (N vs. $M = \lfloor \log_2 N \rfloor$) should make the smart algorithm perform better. This should be especially true when N is quite large so that there is significant difference with its logarithm.

(d) Finally, we should point out that, for most of the cases, the smart algorithm will overcompute powers of A that are not significant (they are equal to 0 for the database concerned). This is not true with the semi-naive algorithm since it computes one power at a time. As N increases the number of joins performed changes only when we cross a power of 2. The smart algorithm should be expected to be particular weak on these points, where the overcomputation is maximal.

To validate our observations above we will apply the formulas for *s_naive_io* and *smart_io* for some specific cases. The number of the parameters involved is significantly large and we will not be able to cover the complete spectrum of possibilities. Nevertheless, we believe that the special cases examined below are enough to give us some insight for the rest also. We assume that our binary relations represent trees. By doing this we avoid worrying about retaining any possible duplicates or not, which is another dimension in the optimization of such operators, yet unrelated to our problem.

We only examine complete trees of outdegree 1 (simple lists), 2 and 3 (see figure 3.5).

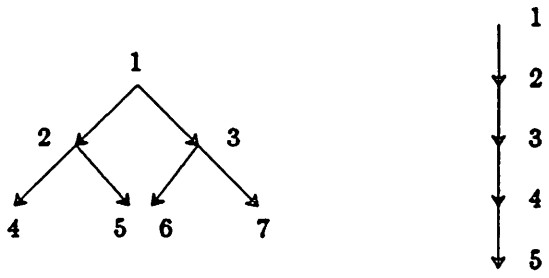

Figure 3.5. Complete trees of outdegree 2 and 1 (list).

We chose to present these categories because they represent two extremes in terms of the ratio of depth over width of the tree. For the various parameters of the problem we chose the following values:

$C = 7$	I/O cost to create a relation
$D = 10$	I/O cost to destroy a relation
$pgsz = 200$	Number of tuples fitting in a page
$buf = 50$	Number of buffers in the system

These values were chosen so that we may compare the results of the formulas derived above with the ones we actually got simulating the semi-naive and smart algorithms on INGRES (see section 3.3). The numbers for C, D and buf where taken as an average of what was observed in a number of experiments. In our experiments we had 8-byte tuples with 2K pages. Due to some overhead information in each page, $pgsz$ was 200. In figure 3.6 we show the plots for the ratio $r = s\_naive\_io/smart\_io$ as a function of the depth of the list/tree, for lists and complete trees of outdegree 2 and 3.

Figure 3.6 validates our comments (a)-(d) above. As the relations grow bigger, sorting becomes more significant and the semi-naive algorithm performs better compared to the smart one. However, we can see that it takes a considerably big relation for this to happen. For lists we need depth at least 2048. Likewise, for complete trees of outdegree 2, the breakpoint depth is 16, which even though it is not a very big number, it corresponds to a considerably big tree of 131072 edges (tuples). Incidently, this relation will need 1Mbyte to be stored in our system, whereas the result for A^* will need approximately 20Mbytes. The same can be said about complete trees of outdegree 3. Figure 3.6c shows that the breakpoint is at depth 8. All the above show that it

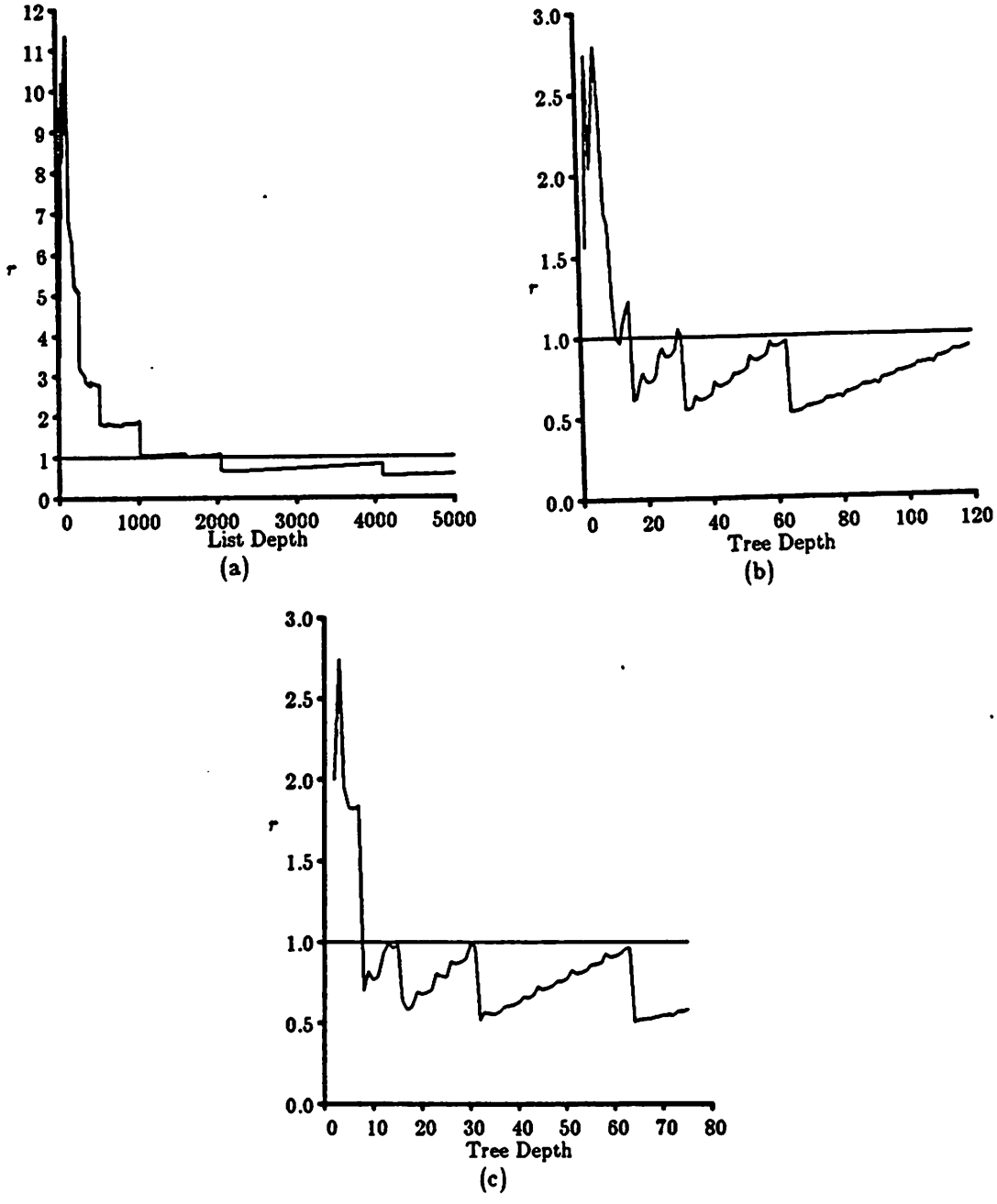

Figure 3.6. Expected relative I/O performance: $r = s\_naive\_io/smart\_io$.
(a) Lists, (b) Complete trees, outdegree 2, (c) Complete trees, outdegree 3

takes an exceptionally big (deep/wide) relation for the semi-naive algorithm to perform less I/O than the smart one. Furthermore, the wider the tree is, the shorter it needs to be for this to happen.

It is also very interesting to see the particular behavior of the ratio $r = s\_naive\_io/smart\_io$. In agreement with point (d) above, there are significant jumps in favor of the semi-naive algorithm at depths $N = 2^k$ for any k. This may be noticed for example at depths 256, 512, 1024, 2048 and 4096 for lists. We can also see that at depths 16, 32 and 64 for complete trees of outdegree 2 and 3. However, there is a noticeable difference in the behavior of lists and complete trees in this respect. For lists r remains relatively invariant between A^{2^k} and $A^{2^{k+1}}$ for some k. On the contrary, for both cases of complete trees examined, as the amount of overcomputation decreases (approaching $A^{2^{k+1}-1}$) the relative performance of the smart algorithm improves significantly. For example, for a tree of outdegree 2 and N=16 we have a drop to $r = 0.61$ but then as N grows it rises again to a point where for N=30 it is $r = 1.04$. So, it seems that $r = s\_naive\_io/smart\_io$ is not a monotone function of N at all.

We have already mentioned that the smart algorithm has been independently proposed in [Vald86], where its performance is analyzed in comparison with the semi-naive algorithm as well. It is difficult to accurately compare the results or our analysis with those in [Vald86]. We used merge-scan join always, whereas they mainly used a hash join similar to the one proposed in [DeWi84]. They also assume a very big buffer pool, whereas we do not. In both points above we were mainly constrained by the system we used for the simulations (see next section). However, there is a common observation of both papers that the smart algorithm performs well in many cases.

3.3. Experimental Performance Results

We have mentioned above that we have simulated the semi-naive and smart algorithms using the commercial version of INGRES [RTI84] on a VAX[1] 11/780 running Unix[2] 4.3. Out of all the performed experiments, we present here the I/O results observed for lists and complete trees of depth 2 and 3 in figure 3.7.

---

[1] VAX is a trademark of Digital Equipment Corporation.

[2] Unix is a trademark of Bell Laboratories.

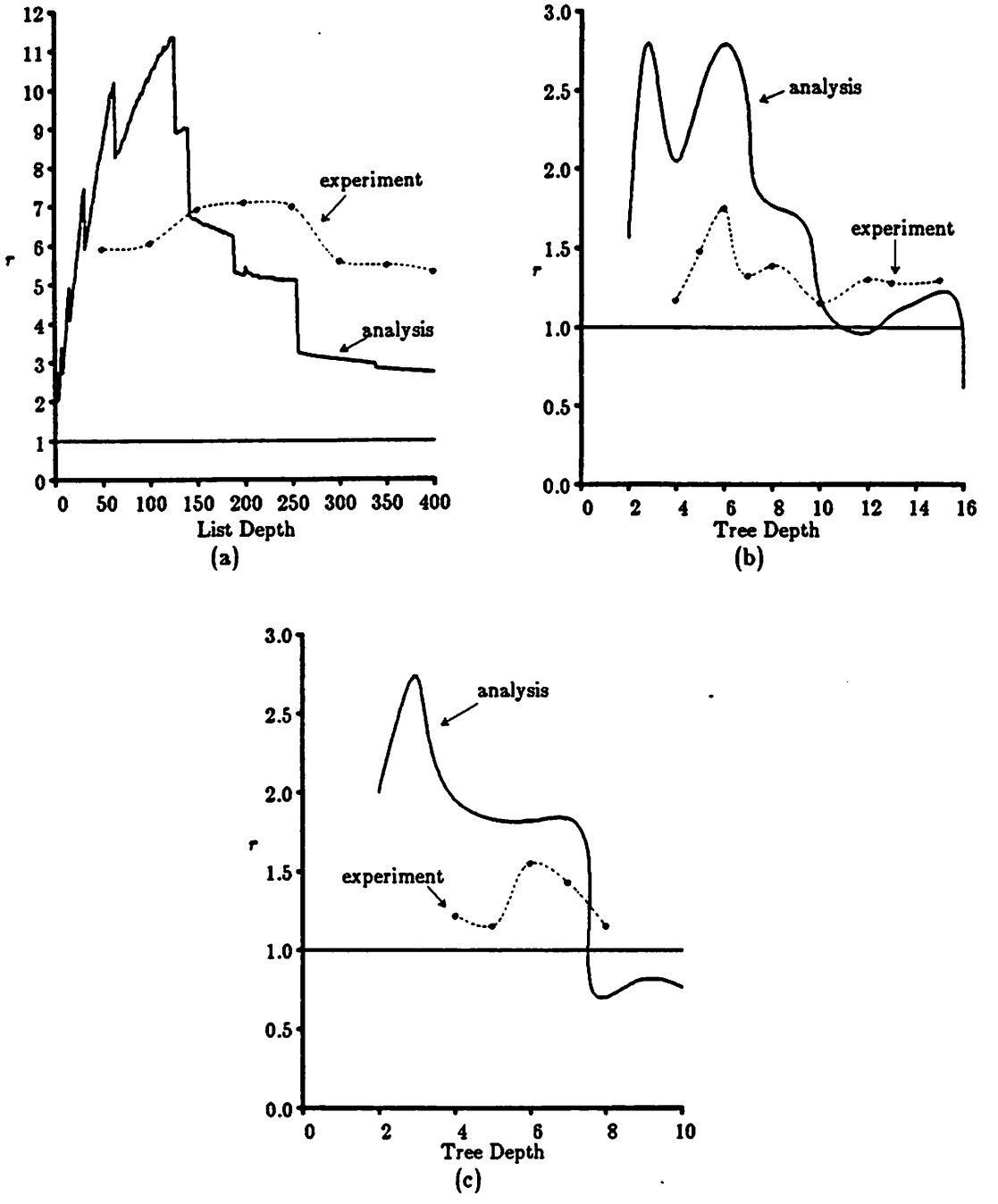

Figure 3.7. Observed relative I/O performance: $r = s\_naive\_io/smart\_io$.
(a) Lists, (b) Complete trees of outdegree 2, (c) Complete trees of outdegree 3

We show again the ratio $r = s\_naive\_io/smart\_io$ together with the corresponding curve given from the analysis of section 3.2. Unfortunately, due to the space (and time) requirements of the experiments we were not able to compare the algorithms on very deep trees, so we were unable to

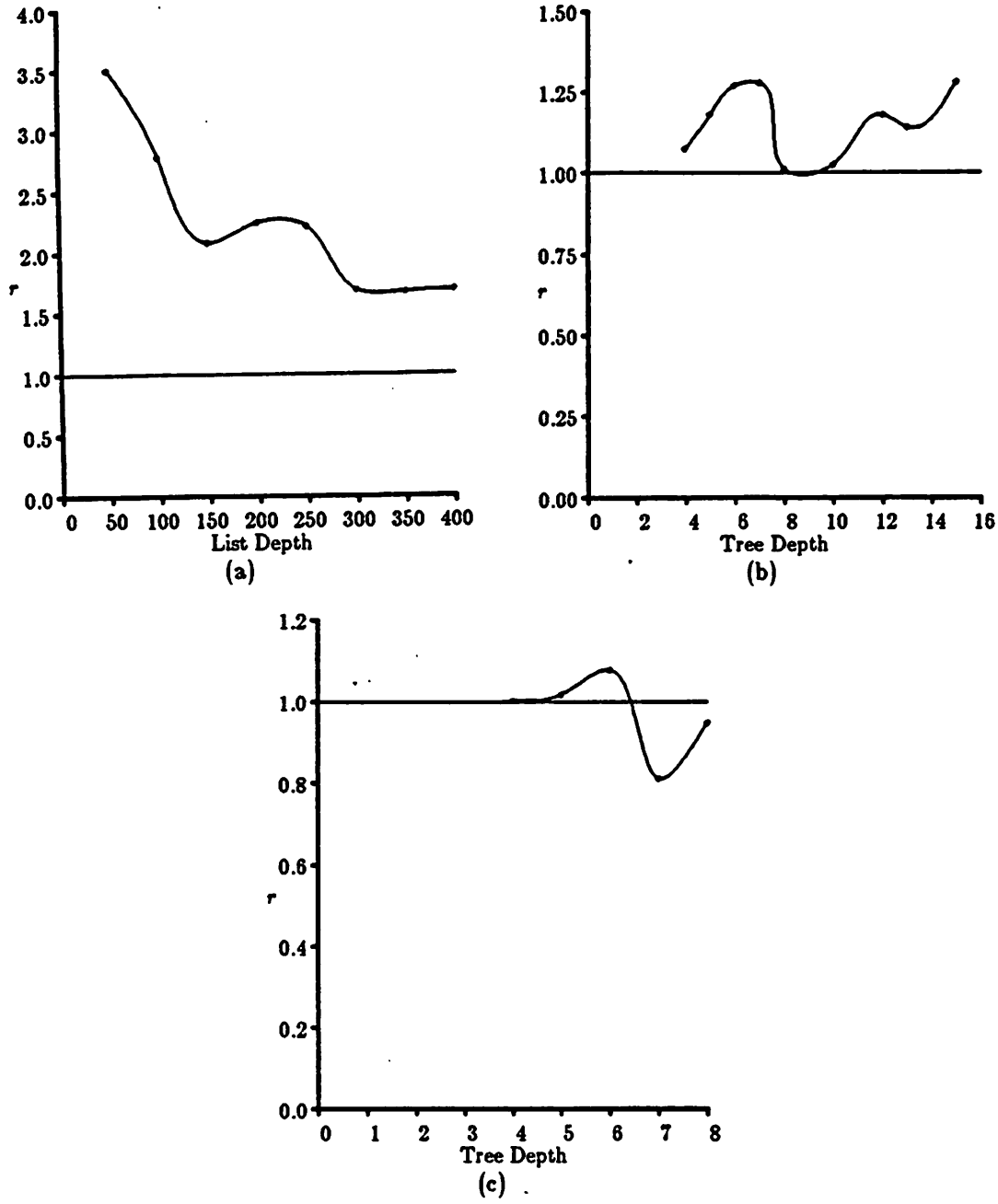

Figure 3.8. Observed relative CPU performance: $r = s\_naive\_cpu / smart\_cpu$.
(a) Lists, (b) Complete trees of outdegree 2, (c) Complete trees of outdegree 3

verify that after some point the semi-naive algorithm becomes better and identify that break-point. Nevertheless, to the extent that we did experiment we see that the results follow more or less what our analysis of section 3.2 showed. Whenever there is a disagreement we believe it is partly because of the simulation overhead and partly because of the pessimism of our model about

the way the optimizer uses the buffer pool and the cost of a sort.

In our experiments we also monitored the CPU time consumed by the algorithms. The ratio $r = s\_naive\_cpu/smart\_cpu$ was the monitored parameter, with s_naive_cpu and $smart\_cpu$ being the CPU time consumed by the semi-naive and smart algorithms respectively. For the same categories of trees as above the observed r is shown in figure 3.8. We can see that the smart algorithm was at least a factor of 2 better in performance for lists, whereas it was from marginally better to marginally worse for trees, for the depths we examined. We can speculate that as the depth of the tree grow, sorting cost will be the significant factor and hence the smart algorithm will behave worse after some point.

4. MORE ALGORITHMS

A natural question, that arises after looking at the smart algorithm (formula (5)), is whether in the same spirit we can come up with other, occasionally even more efficient algorithms. The idea is to look at A^* as a regular expression for which there are many equivalent forms. Formulas (3), (4) and (5) represent only three of these forms and therefore correspond to only three of the possible algorithms to compute A^*. Some possible alternative algorithms can be realized by noticing that A^* may be written as

$$A^* = \cdots (1 + A^{12})(1 + A^6)(1 + A^3)(1 + A + A^2) \tag{6}$$

or

$$A^* = \prod_{k=0}^{\infty}(1 + A^{3^k} + A^{2*3^k}) = \cdots (1 + A^9 + A^{18})(1 + A^3 + A^6)(1 + A + A^2) \tag{7}$$

Soon, we realize that there is an infinite number of ways to write A^*, in the same sense that there is an infinite number of coding systems to code all integer numbers. Testing the performance of a large number of them is prohibitive because of their own time requirements and the complexity of their development. All we can hope for is to follow our intuition and get good heuristics so that the large majority of the suboptimal candidate algorithms are not considered.

Expression (6) differs from (5) in that the grouping starts one iteration later than in (5), that is we compute A and A^2 separately and then we group whatever we got and start applying A^3 and then group everything again etc. On the other hand, expression (7) is even more aggressive in the grouping sense. It iterates twice, before combining everything that it got up to that point and use it in the next iteration.

In terms of the number of multiplications, expression (6) will need approximately $2\log_2 N$, where N is the number of multiplications of (4) (the greater N is, the closer to that number the actual number of multiplications gets). In the same way expression (7) will need about $3\log_3 N$ multiplications. As we can see we may create algorithms that will need $m\log_m N$ multiplications for arbitrary m at the expense of making each multiplication more complex. We may speculate, however, that with a few exceptions $m = 2$ or $m = 3$ at most will be all that we need. In particular, assume that we concentrate on the general form

$$A^s = \prod_{k=0}^{\infty} \left(\sum_{l=0}^{m-1} A^{l*m^k}\right)$$

Formulas (5) and (7) are of this form for $m=2$ and $m=3$ respectively. The number of multiplications required by these formulas is very close to $m\log_m N$. An easy analysis shows that

$$m\log_m N \leq 2\log_2 N$$

only for m in $\{2,3,4\}$. In fact, the expression $m\log_m N$ has a minimum for $m=3$ (restricting m to the integers) independent of N. Therefore, we may immediately conclude that all the other options will be more expensive than (5) and (7) and consider them no further.

Of all the alternative algorithms we chose to experiment only with the one represented by expression (7), which we call the *minimal algorithm* (since it performs the minimum number of multiplications). The analysis in the previous paragraph was the dominant reason for our choice. Of all our experimental results we show here the ones for lists and complete trees of outdegree 2 in figures 4.1a (I/O), 4.2a (CPU) and 4.1b (I/O), 4.2b (CPU) respectively.

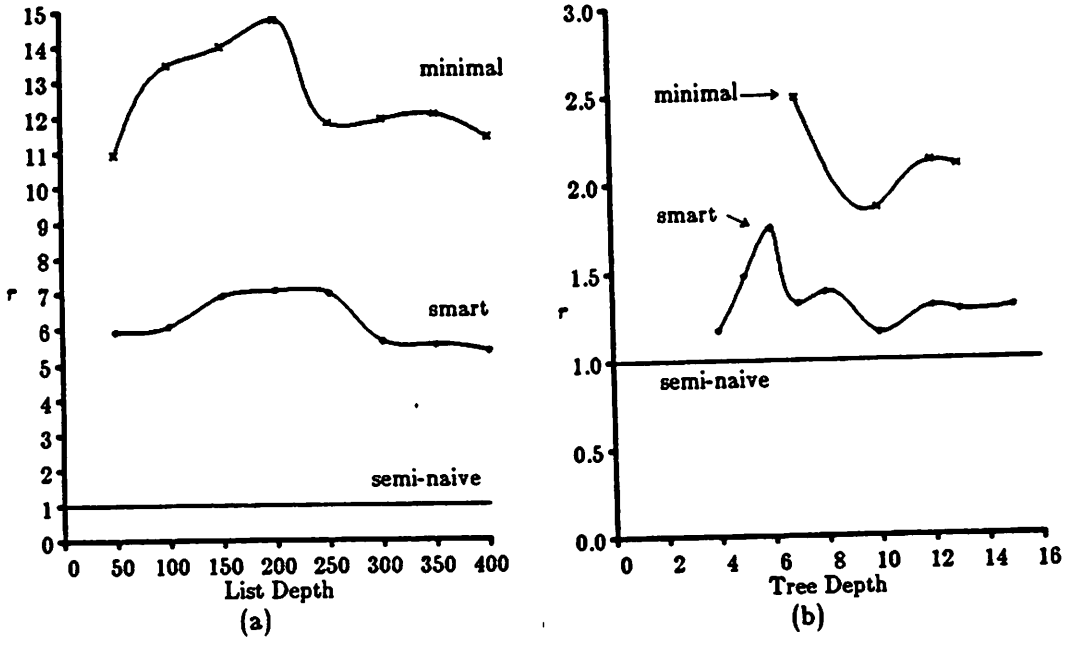

Figure 4.1. Observed relative I/O performance: $r = s\_naive\_io/smart\_io$ or $r = s\_naive\_io/minimal\_io$.
(a) Lists, (b) Complete trees of outdegree 2

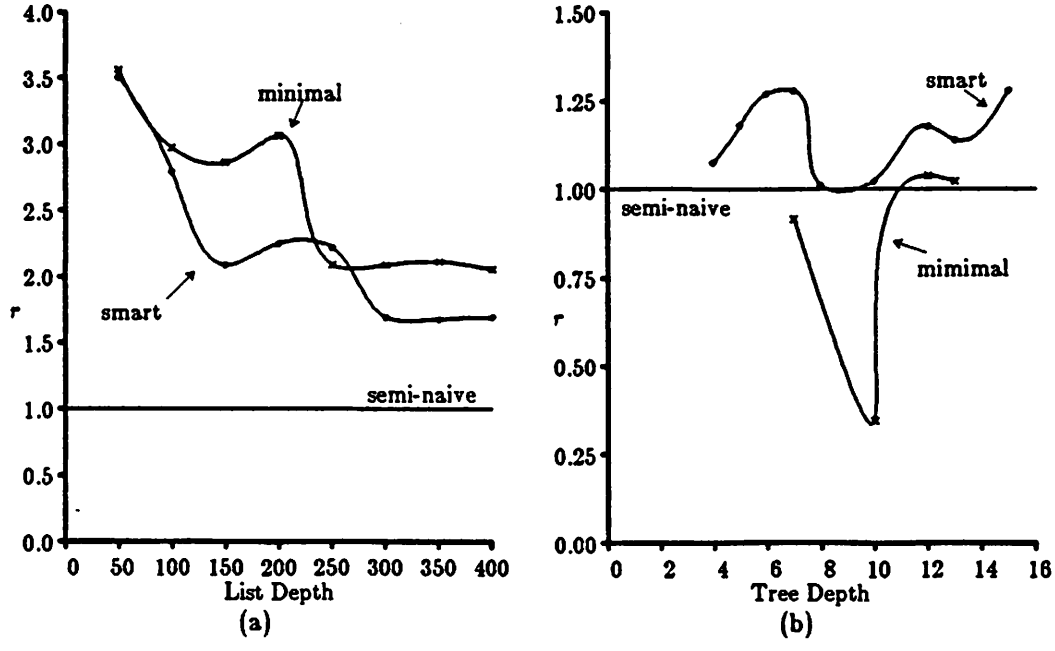

Figure 4.2. Observed relative CPU performance: $r = s\_naive\_cpu/smart\_cpu$ or $r = s\_naive\_cpu/minimal\_cpu$.
(a) Lists, (b) Complete trees of outdegree 2

Assuming that the minimal algorithm consumed *minimal_io* and *minimal_cpu* units of I/O and

CPU time respectively, we are interested in the I/O cost ratio $r = s\_naive\_io/minimal\_io$ and the CPU cost ratio $r = s\_naive\_cpu/minimal\_cpu$. We also show the curves for the corresponding ratios of the semi-naive over the smart algorithm so that all three of them are compared simultaneously.

Figures 4.1 and 4.2 show that the minimal algorithm performs consistently better than the smart one in I/O (and even more so than the semi-naive algorithm). For the range of our experiments it is doing about a factor of 2 less I/O for lists, whereas for complete trees the analogous improvement was about a factor of 1.5. As it concerns CPU performance for lists the minimal was marginally better than the smart algorithm, whereas the opposite was the case for trees. A more extensive set of experimental results is definitely needed to get a better picture of the relative performance of all three algorithms.

5. MORE IDEAS FOR GENERAL RECURSION ALGORITHMS

In the previous sections we have identified and experimented with a small number of equivalent expressions of the power series of a relational operator A, that is A^*. The main characteristic of the expressions that we examined was that A remained unchanged and all we were doing was to find alternative factorizations for the polynomial $\sum_{k=0}^{\infty} A^k$.

One deviation from this, explored in [Ioan86], is to take advantage of the internal structure of A. That is, use the fact that A is (presumably) some composition of more fundamental operators (like join, project etc.) and, having these to be the units of algebraic manipulation, search for equivalent expressions representing more efficient algorithms. For a (still abstract) example assume that $A = BC$. Then we may write

$$A^* = (BC)^* = 1 + B(CB)^*C \tag{8}$$

Expression (8) represents a new algorithm for the computation of A^*, whose significant difference from the original one is the loop on which the iteration is performed, namely $(CB)^*$ instead of $(BC)^*$. Depending on what B and C represent and the contents of the relations that are parame-

ters in B and C, the second algorithm may be much more efficient.

The problem that arises here again is the size of the space of alternative expressions that is thus created. The more complicated the internal structure of A, the bigger the alternative expression space. It is again the subject of our current and future research to see exactly which operators are worth permuting in an attempt to find faster algorithms for A^*.

As a final comment for the computation of A^* we should say that the above alternative algorithms can definitely be combined with the ideas of Sections 3 and 4 about grouping results and decreasing the number of iterations. Taking into account that we have only indicated two or three out of possibly many general ways to find equivalent expressions for A^* and that the two ideas are orthogonal to each other giving us the ability to arbitrarily combine them in any way we want, is definitely showing that any sophisticated optimizer for recursive queries will have a problem of big search space size. In the future good heuristics should be developed to make the search space manageable.

6. CONCLUSION

Using the operator algebra developed in [Ioan86] we have been able to identify many alternative algorithms to materialize recursively defined relations in a database environment. We have analyzed and experimented with a few promising ones, which make a trade-off between the number of multiplications (joins) and their individual costs. Restricting ourselves to computing the transitive closure of trees, our analysis and experimental results have shown that our algorithms perform better than the original one for more shallow relations than for deeper ones. As the relations grow bigger this ceases to be so. However the breakpoint in performance is significantly high in terms of the size (width/depth) of the original relation, which makes the new algorithms more attractive for many of the expected cases. Finally, the results of comparing the two new algorithms examined with each other showed again that, for recursive computations, minimizing the number of multiplications pays off unless the relations are big.

Analyzing and testing the semi-naive, smart and minimal algorithms using join strategies other than merge-scan is part of our current work. Since sorting costs become prohibitive as the relations grow, hash-join techniques seem very promising, as was also pointed out in [Vald86]. We are also planning to investigate the effect of increasing the buffer pool as well as avoiding the creation and destruction of temporaries as much as possible. Finally, it should be interesting to monitor the performance of the three algorithms on more complicated recursive operators than simple transitive closure of binary relations.

In closing we have to comment on the limited scope of the smart and minimal algorithms. Minimizing the number of joins is an issue only when we compute the complete A^*. For queries that involve selections on the underlying relations it is the semi-naive algorithm that should be used taking one step at a time and using the available selections at each point before performing the join. Most likely, this will be much faster than precomputing A^* first (using any algorithm) and applying the selections afterwards. One may validly argue that queries involving selections are much more common than ones asking for a complete materialization of the recursively defined relation. Whether winning in these more rare cases is worth the implementation effort of the smart (or minimal) algorithm is questionable and to some extent application dependent.

Acknowledgements: I would like to give thanks to Prof. E. Wong and to Timos Sellis for all their valuable help.

7. REFERENCES

[Aho79]
Aho, A. and J. Ullman, "Universality of Data Retrieval Languages", in *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, San Antonio, TX, January 1979.

[Banc85]
Bancilhon, F., "Naive Evaluation of Recursively Defined Relations", in *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, FL, February 1985.

[Banc86]
Bancilhon, F., D. Maier, Y. Sagiv, and J. D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs", in *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systemsdf*, Boston, MA, March 1986, pages 1-15.

[Baye84]
Bayer, R., "Query Evaluation and Recursion in Deductive Database Systems", in *Proceedings of the Islamorada Workshop on Knowledge Base Management Systems*, Islamorada, Florida, 1984.

[Blas76]
Blasgen, M. W. and K. P. Eswaran, *On the Evaluation of Queries in a Relational Data Base System,* Research Report RJ-1745, IBM San Jose , April 1976.

[DeWi84]
DeWitt, D. J., R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems", in *Proceedings of the 1984 ACM-SIGMOD Conference*, Boston, MA, June 1984.

[Ende72]
Enderton, H. B., *A Mathematical Introduction to Logic,* Academic Press, New York, N.Y., 1972.

[Gall84]
Gallaire, H., J. Minker, and J. M. Nicolas, "Logic and Databases: A Deductive Approach", *ACM Computing Surveys* **16**, 2 (June 1984).

[Gutt84]
Guttman, A., *"New Features for Relational Database Systems to Support CAD Applications"*, PhD Thesis, University of California, Berkeley, CA, June 1984.

[Han86]
Han, J. and H. Lu, " Some Performance Results on Recursive Query Processing in Relational Database Systems", in *Proceedings International Conference on Data Engineering*, Los Angeles, CA, January 1986, pages 533-539.

[Hens84]
Henschen, L. and S. Naqvi, "On Compiling Queries in Recursive First-Order Databases", *JACM* **31**, 1 (January 1984).

[Ioan86]
Ioannidis, Y. E. and E. Wong, "An Algebraic Approach to Recursive Inference", in *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986.

[RTI84]
RTI, *INGRES Reference Manual, Version 2.1,* July 1984.

[Rose86]
Rosenthal, A., S. Heiler, U. Dayal, and F. Manola, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", in *Proceedings of the 1986 ACM-SIGMOD Conference on the Management of Data,* Washington, DC, May 1986.

[Ullm85]
Ullman, J., "Implementation of Logical Query Languages for Databases", *ACM TODS* **10**, 3 (September 1985), pages 289-321.

[Vald86]
Valduriez, P. and H. Boral, "Evaluation of Recursive Queries Using Join Indices", in *Proceedings of the 1st International Conference on Expert Data Base Systems,* Charleston, SC, April 1986, pages 197-208.

[Viei86]
Vieille, L., "Recursive Axioms in Deductive Databases: The Query / Subquery Approach", in *Proceedings of the 1st International Conference on Expert Data Base Systems,* Charleston, SC, April 1986, pages 179-193.