A Comparison of I/O Performance of Some Linear Recursive Query Processing Methods

W.S. Luk and Simon Mok
School of Computing Science
Simon Fraser University
Burnaby, B.C. Canada V5A 1S6

Abstract

This paper studies the I/O performance of four popular query processing methods for deductive databases when they are applied to a linear recursive rule set popularly known as the same-generation rule set. These methods are: Henschen-Naqvi, Counting, Reverse Counting and Magic Set. Our analysis and simulation identify Counting as the most efficient method so long as there is not an extraordinarily deep recursion. The paper contains also a new characterization of these four methods according to the ordering in which the necessary join operations are performed. The scheme we adopt to present these methods is arguably more natural, and hence more easily understandable than the scheme used to present them in the original paper. Since it is more implementationally oriented, it also provides a better framework for performance comparisons.

1. Introduction

According to the general convention ([GMN 84]), a deductive database (DBD) consists of two components: intensional and extensional databases (IDB and EDB). Deductive rules are stored in the IDB while the elementary facts are stored in EDB from which new facts are derived by applying the deductive rules. Here, the IDB is considered to correspond to the database in an ordinary, non-deductive DBMS. A class of deduction rules, called linear recursive rules, has recently attracted a lot of attention from many database researchers ([HN 84], [BR 86], [HL 86], [YZ 87] & [HH 87]). In this paper, we will study the query processing strategies for a typical example of linear recursive rules which is designed to solve the so-called same-generation problem ([BMSU 86]). This rule set is:

Example (Rule) 1:

\[
\begin{align*}
p(X,Y) & : \text{UP}(X,W), p(W,Y), \text{DOWN}(Y,Y). \\
p(X,Y) & : \text{FLAT}(X,Y). \\
\text{query}(X) & : p(a,X).
\end{align*}
\]

where UP, FLAT and DOWN are base relations. a in the last rule is considered to be a vector of constants.

There are already many good linear recursive query processing algorithms. Three of them are frequently cited: Henschen-Naqvi (HN), Counting (CN) and Magic Set (MS) ([HN 84], [BMSU 86] & [BR 86]). MS is an algorithm that can be applicable to non-linear cases, while CN is applicable only when the data is non-cyclic. There have been some performance studies to compare the algorithms, all of which are based on the linear recursive rule set in Example 1. [BMSU 86] studied the complexity of the algorithms based on specific databases. A formal study was attempted by [BR 86] which measured the sizes of the intermediate relations generated during the query processing. Against a set of parameter values representing a database, the size of the intermediate results (before duplicate removal) generated by each algorithm under evaluation was estimated analytically. [HL 86] adopted a more experimental approach to performance evaluation of linear recursive query processing algorithms. They studied a class of algorithms that can be characterized as 'wavefront' algorithms. (Of the three algorithms mentioned above, HN is one of them while CN and MS are not.) The cost formula included CPU and I/O cost components. The I/O cost was calculated on the basis of the estimated number of index and data pages to be accessed (i.e., page I/O). Their findings were confirmed by the actual processing times of the algorithms against randomly generated databases.

The focus in this paper is on disk I/O performance, although page I/O will also be studied. All four algorithms that were studied in [BMSU 86], are included here (i.e., CN, MS, HN and RN (Reverse Counting)). Since all but HN are actually algorithms to re-write the recursive rules, each of these algorithms is applied to Example 1 to transform it into another equivalent set of recursive rules. These new recursive rules, as shown in [BR 86] are much more specialized and considered to be specific query processing methods. Therefore, for the purpose of this paper, the algorithm and the corresponding query processing method for Example 1 are treated to be the same thing, unless otherwise specified.

We will analyse the disk I/O performance of each method by studying the ordering, according to which the necessary join operations are performed. This ordering is illustrated by the pattern in which the three base relations, i.e., UP, FLAT and DOWN, are accessed. Since it is difficult to estimate quantitatively the disk I/O performance because of the buffering effect, we resort to simulation to provide some empirical evidence to support our analysis. To obtain the measurements of page I/O and disk I/O, we implement and run these methods against randomly generated databases. From the recorded measurements we may compare their I/O performance.

Acknowledgement: This research was supported by research grants from the Natural Sciences Research Council of Canada and the Centre for Systems Science, Simon Fraser University.

International Symposium on Database Systems for Advanced Applications
Seoul, Korea, April, 1989

- 293 -
2. Recursive Algorithms

Recursive rules are also called logical rules, reflecting their Prolog origin. In the literature, recursive rules are invariably expressed as Horn clauses with no function symbols. Thus, it is not surprising that most of strategies proposed for evaluating recursive queries aimed at improving Prolog-style processing in the database environment. For example, in [BM8U 89], the Magic Set method was proposed as a bottom-up method with a polynomial time bound (as opposed to top-down method with an exponential time bound adopted by Prolog). Counting and Reverse Counting were proposed as specialized versions of Magic Set method to deal with the same-generation problem, i.e., the Example 1. In this paper, we take a more implementational view of recursive query processing. Essentially, we consider the task of query processing as one to pick an ordering according to which a set of join/selection operations are performed.

We first convert the Rule into an iterative scheme as follows:

Level 0 : \( \sigma_{FLAT} \)
Level 1 : \( \sigma_{FLAT} \ast DOWN \)
Level 2 : \( \sigma_{UP}\ast FLAT \ast DOWN \ast DOWN \)

where \( \sigma_i \) denotes the selection of the relation (i.e. UP or FLAT) and * denotes the natural join of two relations. The method will terminate at the ith level of iteration if the result obtained at that level processing is no new tuples when compared with the results obtained at the (i-1)th level. The selection selectivity value of \( \sigma_i \) is proportional to the number of tuples in the result relation, and the join selectivity has a similar meaning for a join operation. More formal definitions of these selectivities will be given in Section 3. Here the values of these selectivities indicate the volume of data that need to be handed.

The four methods chosen by this work can be categorized by whether they are stage-first or level-first and whether they are top-down or bottom-up.

We partition the set of all join operations in the above scheme into three stages:

1. Up-stage: the set of all join operations involving the UP relation
2. Flat-stage: the set of all join operations involving the FLAT relation and intermediate relations that have been derived from operations in the Up-stage
3. Down-stage: the set of all join operations involving the DOWN relation and other intermediate relations that have been derived from the operations in the Up-stage and Flat-stage.

An method is stage-first if all the join operations in one-stage are performed before any join operation in another stage. Usually, the operations in the Up-stage come first, with the result at each level saved in an intermediate relation. Then these relations are joined with the FLAT relation, creating another set of intermediate relations which are then joined with the DOWN relations. In contrast, an method is level-first if the join operations at each level are performed before any join operation at next level. As an analogy in matrix operations, a stage-first method proceeds with elements in the matrix (i.e. join operations) by columns (i.e., stages) while a level-first method proceeds by rows (i.e., levels).

The top-down approach is distinguished from the bottom-up approach in that the former opts for the selection to be done before any join operation in the Up-stage. In contrast, the bottom-up approach collects all tuples that are possible candidates for inclusion into the answer to the query. Then the selection is applied to yield the exact answer.

There are now four methods, which are the combinations of two factors: (a) stage-first or level-first, and (b) top-down or bottom-up.

We will analyze each method in turn:

1. Hanssen-Naqvi (HN):

   This is a top-down, level-first method.

   \[
   \begin{align*}
   \text{Level 0} & : \sigma_{FLAT} \\
   \text{Level 1} & : \sigma_{UP}\ast\sigma_{FLAT}\ast DOWN \\
   \text{Level 2} & : (\sigma_{UP}\ast\sigma_{FLAT})\ast DOWN \ast DOWN \\
   \text{Level 3} & : (\sigma_{UP}\ast\sigma_{FLAT})\ast DOWN \ast DOWN \ast DOWN
   \end{align*}
   \]

   where \( \sigma_{UP} \) is an intermediate relation that has been computed at the previous level (i.e. \( \sigma_{UP} = \sigma_{UP-1}\ast\sigma_{UP} \)).

   This method requires that, for each level of processing (except for the first one), the base relations UP, FLAT and DOWN be accessed once and in that order. If there are no indices on the join attributes of these relations, all tuples of each base relation must be accessed at least once for each level of processing. Even with the provisions of all these indices, if the values of the selection selectivity of \( \sigma_i \) and the join selectivities between any pair of the base relations are large, a large portion of each base relation will still be accessed once for each level of processing. For a small buffer size which cannot hold all the basic relations in memory, there will be heavy disk traffic since pages of all the base relations must be brought into memory for processing at each level.

2. Counting (CN):

   This is a top-down, stage-first method as shown in Table 2-1.

   It is clear that this method has a much better reference locality than all other methods, when the number of levels involved is not too large. All the base relations are processed one at a time. Once the processing is over for a stage, the base relation will no longer be required for the rest of the processing. As a stage-first method, the base relation processed in one stage will be represented by a series of intermediate relations during the next stage of processing, e.g. the base relation UP is represented by the series of intermediate relations. It is important that these intermediate relations must not be very large, otherwise a large buffer is required to avoid heavy disk traffic brought about by swapping and then reading these intermediate relations again. Note that HN, as a level-first method, does not have to store away any intermediate relations for processing at next level.

3. Magic Set (MS):

   This is a bottom-up, level-first method which includes a preprocessing to compute the magic set.

   Pre-processing:

   \[
   \begin{align*}
   \text{Level 0} & : M_0 = \sigma_{UP} \\
   \text{Level 1} & : M_1 = M_0 \ast \sigma_{UP} \\
   \text{Level 2} & : M_2 = M_1 \ast \sigma_{UP} \ast \sigma_{FLAT} \\
   \text{Level 3} & : M_3 = M_2 \ast \sigma_{UP} \ast \sigma_{DOWN} \\
   \text{...} & : MAGIC = \sum_{i=0}^{5} M_i \\
   \end{align*}
   \]

   **MUP** = MAGIC \* UP

   **MFLAT** = MAGIC \* FLAT

   \[
   \begin{align*}
   \text{Level 0} & : M_{F0} = MFLAT; \sigma_{MF0} \\
   \text{Level 1} & : M_{F1} = MUP \ast M_{F0} \ast \sigma_{MF1} \\
   \text{Level 2} & : M_{F2} = MUP \ast M_{F1} \ast \sigma_{MF2} \\
   \text{Level 3} & : M_{F3} = MUP \ast M_{F2} \ast \sigma_{MF3}
   \end{align*}
   \]

   After the pre-processing, the base relation UP is replaced by a hopefully smaller intermediate MUP, and the base relation FLAT is replaced by a series of intermediate relations MF. The whole
To sum up, we have the following comparisons:

- The base relations UP and FLAT are used instead of MUP and MFLAT at each level of processing.
- The idea of magic set is to eliminate tuples belonging to UP and FLAT that clearly do not contribute to the answer of the query. Thus, the pre-processing will help only when the magic set can filter out a significant portion of these non-contributing tuples, which in turn depends on the selectivity of the selection \( s \).
- Nonetheless, there are situations in which this magic set method achieves very good I/O performance. When the value of the selection selectivity is small, all intermediate relations will also be correspondingly small. Thus, the magic set method could be as efficient as counting method.

4. Reverse Counting (RC):

   It is a bottom-up, stage-first method. It goes through the same pre-processing as MS. In addition, the intermediate binary relation \( MU \) is computed such that its two columns are equal to the first column of MFLAT.

   \[
   \begin{align*}
   \text{Up-Stage} & \quad \text{Down-Stage} & \quad \text{Flat-Stage} \\
   \text{Level 1} & \quad X_1 = \mu^*p^*z_1 & \quad \sigma_\varphi(x_1\times z_1) \\
   \text{Level 2} & \quad X_2 = \mu^*x_1 \quad z_2 = \sigma_\varphi(x_2\times z_2) \\
   \text{Level 3} & \quad X_3 = \mu^*x_2 \quad z_3 = \sigma_\varphi(x_3\times z_3) \\
   \end{align*}
   \]

   

   This method will perform reasonably well only if the join selectivity value of FLAT and DOWN is kept very small. Otherwise, it would not compete with Counting or Magic Set. In comparison with Counting, it has a low degree of reference locality because both series of intermediate relations \( X_i \) and \( Z_i \) must be accessed again during the final stage. In comparison with Magic Set, the join operations in the final stage really slow down the Reverse Counting method. What hurt most is that each relation involved in the series of join operations is unique, unlike the join operations in all other methods where an index can be built for one relation for the entire series of join operations.

   In fact the Reverse Counting method is so inefficient that given our limited computing resources, we had a difficult time to run this method in our simulation study as described below. As a result, we had to drop this method in the rest of this paper.

To sum up, we have the following comparisons:

1. **Stage-first vs. Level-first:**

   - Stage-first approach is usually more efficient than the level-first approach since the buffer requirement for the former is limited to
   
   - the size of the base relation and its index. For the level-first approach, if the buffer size falls short of total size of three base relations, pages of these base relations will be brought in and out of memory for each level of processing, as stipulated by the LRU page replacement policy.

   One problem of the stage-first approach which is not shared by the level-first approach is that at each stage of processing, a series of intermediate relations is generated which must be stored and read again for the next stage of processing. Thus, if there are many levels of iteration, there will be heavy disk traffic for stage-first methods.

2. **Top-down vs. Bottom-up:**

   Top-down approach is generally more efficient than the bottom-up approach for the reason that selection before join is generally a better strategy than join before selection. In addition, the bottom-up has a larger storage requirement for the intermediate relations for two reasons. First, the top-down approach generates unary intermediate relations since the answer will be a unary relation. The bottom-up approach however requires binary intermediate relations since the selection is postponed until the end. Second, the bottom-up approach generates redundant tuples that contribute to the derivation of the answer.

   - Pre-processing is a strategy that will hopefully reduce the number of redundant tuples before the join operations take place. This strategy may help if the filtering process is very effective, i.e., the magic set is small. There are two problems with pre-processing. First, the cost of pre-processing must be taken into account. Second, and potentially more serious one, there are many intermediate relations generated as surrogates for the base relations. These relations will be joined with themselves. These join operations will be more expensive than the join operations in the top-down approach which need to build indices for only the base relations.

3. **Simulation:**

   This section describes the details of the simulation which produces simulation data for analysis in Section 4. The reader who are not interested in the simulation details may skip this section.

   In order to study comparatively the I/O performance of these methods which involves the effect of buffering, we have built a program (package of programs) to simulate the database architecture which will be described in Section 3.1. The rest of this section is to provide more specific details about the simulation program.

3.1. **Architecture of the Database System**

   The system consists of two sub-systems: Relational Sub-system (RS) and Data Storage Sub-system (DSS) (see Fig. 2.1).

   The primary task of RS is to execute various relational operations as required by the database program. Data relations are accessed, in a tuple-at-a-time fashion, when a relational operation is being executed. Therefore the interface between the RS and DSS is a tuple-interface. Input commands from RS to DSS are mostly of tuple-read and tuple-write type. When a tuple is required/inserted during the execution of a relational operation, a sequential tuple-read/write or a direct tuple-read/write (with key) request is made.

   The DSS is further decomposed into two modules: File Manager (FM) and Buffer Manager (BM). The major function of FM is to convert a tuple request issued by the RS into a page request which is then transmitted to BM. It also manages indices (i.e., B-trees) to base
The BM manages a buffer space, the size of which is measured by number of pages. BM converts a page request issued by the FM into a request for this page from the disk, unless the page requested happens to be found in the buffer. If a page is retrieved from the disk to be placed into the already full buffer, one of the existing pages must be replaced. In this case, the BM will according to a page replacement policy pick a page to be overwritten by the new page.

There are many page replacement policies for buffer management. However, performance comparisons of various policies are outside the scope of this study. Thus, the ubiquitous LRU (Least Recently Used) is chosen as the page replacement policy. This choice will focus our discussion on the effect of buffering on the methods. To reduce the running time of the program that simulates this model, we assume that the method will request, via a FREESPACE command, the BM to release the buffer space occupied by (base or intermediate) relations which will no longer be accessed later on.

3.2. Linear Recursive Query Processing Algorithms

We choose CN, HN and MS as the candidate methods for comparison. The programs they produce for the set of linear recursive rules as shown in Example 1 are shown in the pseudo-code format in the Appendix A. In this paper, the performance of these methods are actually taken to be the performance of these programs.

To reduce the simulation time, each program runs for at most five iterations (levels). For randomly generated databases, this restriction should not affect the generality of the simulation results. In Section 5, we will consider the performance of these methods when they are executed against specific databases, without this restriction.

Each program executes a number of relational operations, which are the sources for page I/O and consequently disk I/O activities.

3.3. Parameters of Simulation Model

This subsection discusses the parameters chosen as variables in the simulation model. There are four parameters: Join Selectivity (JS), Selection Selectivity (SS), buffer size and page size. The first two are related to the volume of data that will be processed recursively. They will be further elaborated in next subsection on generation of simple databases. The values chosen for the other two parameters are as follows:

- Buffer size: 10, 25, 50, 75 and 150 pages
- Page size: 20 and 40 tuples/page

3.4. Database Generation

Each of the three base relations (UP, FLAT and DOWN) contains 1000 tuples, each of which consists of 2 integers. Each integer in a column is chosen randomly over a given range, independently of the integers in another column. However, performance comparisons of various policies are outside the scope of this study. Thus, the ubiquitous LRU (Least Recently Used) is chosen as the page replacement policy. This choice will focus our discussion on the effect of buffering on the methods. To reduce the running time of the program that simulates this model, we assume that the method will request, via a FREESPACE command, the BM to release the buffer space occupied by (base or intermediate) relations which will no longer be accessed later on.

Selection Selectivity (SS) depends directly on size of constant vector in the query. If it is a m-tuple vector, the selection selectivity of this query is 1/1000. Six SS values are chosen: 0.001, 0.005, 0.015, 0.100, 0.300 and 0.500.

3.5. Execution of Simulation Program

For each distinct set of values of three parameters (JS, SS and page size), the simulation program is executed five times. Each time a different set of base relations are generated, by means of a different seed for the random number generator. All three methods are run against the same database for all buffer sizes, and all relevant data are recorded, which include the page I/O and disk I/O activities, as measured by number of pages. Then the average values over the five runs are calculated.

Besides the randomly generated databases, we also run the methods against some specific databases which have been designed to magnify the differences among the three methods (see Section 5).

4. Analysis of Simulation Data

The simulation data are charted in Figure 4-1, where the page size is 20 and join selectivity = 0.001. Due to space limitation, we do not show the simulation data for the other two selectivity values: 0.005 and 0.0001. Suffices it to state that they follow similar trend as shown in Figure 4-1. Nor do we include those data when page size is 40 (tuples per page), because they are very similar to their counterparts shown in the figures. The only difference is that when the page size is 40, the disk I/O performance of the methods tends to converge to a steady state at a point where the buffer size is only half as large.

4.1. The Winner Is

It is very clear that CN is the best method for all combinations of values of the four parameters we consider here. Of the four, the most interesting parameter to watch is the buffer size. For all values of SS and JS, the disk I/O performance of CN improves very rapidly as the buffer size increases from 10 to 75 pages, then it starts to level off.
Our analysis of CN in Section 2 showed that at each stage of processing, only one base relation (size = 50 pages) is processed. Thus what the simulation data indicate is that when the buffer size is large enough to hold all the pages of the base relation and the associated index pages in memory for all join operations for that stage, additional buffer memory does not help much. This seems quite a surprising result, given that there are other intermediate relations involved in the join operations. Our explanation is as follows. When the selectivity values are very small, these intermediate relations are very small and can fit into the buffer, so that there will be no disk I/O incurred except for the necessary initial disk I/O to bring all the pages into the buffer. If intermediate relations are large, each of them will be created and written on the disk soon after. This is because each operation involves the base relation and an intermediate relation that will not be referenced again until the next stage. According to the LRU buffer management...
scheme, the base relation will always be memory resident, while the
intermediate relations will be moved to disk one by one. Thus,
unless the buffer size is increased to the point where all intermediate
relations may be saved in buffer for next stage of processing,
increase in buffer size will not provide much help in reducing disk
I/O. This point will be observed again in Section 5.
It is not easy to determine whether HN is better than MS in overall
disk I/O performance. HN is definitely better than MS for small
buffers. This is largely because the page I/O of HN is less than that
of MS, according to Table 4-1. When the buffer size is small, the
buffering effect is quite small and the disk I/O can be approximated
by the page I/O. On the other hand, the disk I/O performance of MS
is more responsive to the increase in buffer size. We can explain this
by considering our analysis in Section 2. While both MS and HN are
level-first methods, HN scans all base relations for each level of
processing and MS scans the smaller surrogates of UP and
DOWN. In fact, for small SS values when the sizes of the surrogates
are minimal, the disk I/O performance of MS is very close to that of
CN. On the other hand, for large SS and JS values, MS fares the
worst for all buffer sizes.

4.2. Page I/O vs. Disk I/O
The Table 4-1 shows the page I/O of each method as the Selection
Selectivity (SS) varies. Comparing Table 4-1 with Fig. 4-1, one can
readily see that the comparative page I/O performance strongly
influences the comparative disk I/O performance, when the buffer is
very small. This shows that for DBMS with small buffer size, disk
I/O can be approximated by page I/O, for comparison purposes.
When the buffer size increases slightly, in terms of the size of the
base relation, reference locality becomes a significant factor in the
disk I/O performance. For example when buffer size is 25 pages
(i.e., one fourth of the size of a base relation) MS has better
performance than HN for small selectivity values (i.e., 0.001 and
0.005).

The three methods have similar page I/O performance when the
selectivity values are small. When the selectivity value increases, the
page I/O of MS increases much faster than the other two, primarily
because of the size of its intermediate relations which are binary,
instead of unary in the other two cases. What is surprising is that CN
and HN are very close in their page I/O performance, as Table 4-1
shows. Evidently, the superior disk I/O performance of CN over HN
is largely due to the way the base relations are accessed by each
method.

5. Vulnerability of the Counting Method
Counting (CN) has been shown to be the best method for all
combinations of parameter values tested by our simulation
experiments. Of course, the one restriction that each method runs at
most five iterations seems to put CN, a stage-first method, in a more
favorable position relative to the other two which are level-first
methods. However, we found no discernible differences when we
repeated some experiments where the methods were allowed to run
two more iterations. HN scans all base relations for each level of
processing and MS scans the smaller surrogates of UP and
DOWN. We suspect that in the increase in the number of iterations alone will not affect its
disk I/O performance. The intermediate relations of CN have to
be small enough so that they must be taken into account just as the
MUF and M2 in the case of MS. We conjecture that if there are a
large number of intermediate relations at each stage (i.e., a deep
recursion, each of which is quite large, then CN, as a stage-first
method will lose out to level-first methods such as HN and MS.

5. Selectivity CN HN MS
---
<table>
<thead>
<tr>
<th>S. Selectivity</th>
<th>CN</th>
<th>HN</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.001</td>
<td>11979</td>
<td>12000</td>
<td>12157</td>
</tr>
<tr>
<td>0.005</td>
<td>11969</td>
<td>11995</td>
<td>12152</td>
</tr>
<tr>
<td>0.050</td>
<td>15608</td>
<td>15648</td>
<td>19164</td>
</tr>
<tr>
<td>0.100</td>
<td>18041</td>
<td>18119</td>
<td>23490</td>
</tr>
<tr>
<td>0.300</td>
<td>24651</td>
<td>25676</td>
<td>44362</td>
</tr>
<tr>
<td>0.500</td>
<td>28721</td>
<td>30021</td>
<td>53085</td>
</tr>
</tbody>
</table>

Table 4-1: Page I/O Comparison in Page Requests
(JS=0.001, 20 tuples/page)

Interestingly, our simulation experiments against the same example
databases produce quite different results, again due to the effect of
buffering.

• Example (database) 2:
  UP: \((a_1, a_1), 1 \leq i \leq n\).
  FLAT: \((a_1), 3 \leq i \leq n\).
  DOWN: \((b_1, b_{i-1}), 2 \leq i \leq n\).

We choose a value of 25 for \(n\). This means the processing will
continue until the 25th level. The linear recursive rule in Example 1 is applied to this database with query \(p(i, i, 1)\).

Some observations we made for randomly generated databases are
also applicable here. For example, HN and MS do not perform as
well as CN for the small buffer size. As the buffer size increases, MS
outperforms the other two for the reason pointed by [BMSU 86],
amely, asynchronous data. The surprise here is that as buffer
size increases to about 11 pages, HN, which suffers from the same
problem as CN, starts to attain its optimal performance, when all
base relations being accessed at each iteration can be accommodated by the large buffer space. CN, however has to
store all intermediate relations derived at all 25 levels, so that its
buffer requirement for efficient processing is much larger than 11
pages. As a result, CN virtually does not respond to any moderate
increase of buffer size, a point which has been explained in
Section 4.1.

• Example (database) 3:
  UP: \((a_1, a_1), 1 \leq i \leq n\).
  FLAT: \((a_1), 2 \leq i \leq n\).
  DOWN: \((b_1, b_{i-1}), 2 \leq i \leq n\).

MS turns out to be the best method to process this database. (In
fact, the optimal method is Semi-Naive ([BR 86]) which quits
after the first iteration.) The comparative performance of HN and
MS seems to be identical to that for Example 2. This however is
due to the fact that according to our implementation, the
intermediate relation generated by CN at each iteration (level) is
stored in a separate page, in order to maintain the ady of this
intermediate relation (= 1). For this example, each intermediate
relation generated by CN at each level consists of only one tuple.
Thus it is the internal page fragmentation that causes poor
performance of CN. By compacting all intermediate relations
Together, the modified CN (called Compacted CN) outperforms
HN. Note that this Compacted CN does little to improve the
performance of CN in the case of Example 2.

6. Conclusion
Our analysis and simulation show that Counting is the best in disk
I/O performance under all circumstances studied in this work,
however, against databases with asynchronous data, Counting will
fail miserably. The Reverse Counting, even though it was devised
using the same rationale as Counting, is the worst method. It is a
toss-up between Henschen-Naqvi and Magic Set as each has its
comparative strengths and weaknesses. In general, Magic Set does
well for small selectivity values, especially when the buffer is not too
efficient. With respect to page I/O, our simulation data show that
Counting has the best performance, although Henschen-Naqvi has
almost the same performance. The page I/O performance of Magic Set
deteriorates rapidly as the value of selection selectivity increases.
Given that Magic Set is considered to be a safe method, as defined in
[Z 86], and all other three methods will not handle cyclic databases,
it seems that the choice for a strategy to process Example 1 is
between Counting and Magic Set.

One contribution of this research has to do the way in which each
method is presented and analyzed. Both the original paper
describing the Magic Set, Counting and Reverse Counting [BMSU
86] and the tutorial paper [BR 86] have very different ways to
characterize these algorithms. In our opinion, the characterization of
these four algorithms according to whether an algorithm is stage-first
or level-first and whether it is top-down or bottom-up, is more
natural, hence more easily understandable. Moreover, in this way, it
is easier to compare, not only just disk I/O performance, but also
page I/O performance and processing requirements in general (e.g.
the need for indexing for join operations), and as a result, easier to
account for the differences in performance of any two methods.
Besides the tutorial value of our presentation scheme, we hope it will
provide a useful framework of designing recursive query processing
algorithms in general. It is our conjecture that the distinction
between stage-first and level-first is an important one for other
classes of recursive query processing algorithms that require
accessing several relations repeatedly.

7. Acknowledgement
This research was supported in part by the Natural Sciences and
Engineering Council of Canada and the Center for Systems Science
at SFU.

Appendix A: Pseudo-Codes of the Algorithms

**CN**

Set $C_1 = \text{Project(Select(UP))}$
For $i=2,3,4,5$ Do
Begin
   $C_i = \text{Project(Join}(C_{i-1},UP))$
End
For $i=5,4,3,2,1$ Do
Begin
   $F_1 = \text{Project(Join}(C_1,FLAT))$
End
Set $D_5 = F_5$
For $i = 4,3,2,1$ Do
Begin
   $T_i = \text{Join}(D_{i+1},\text{DOWN},F_i)$
End
Set $A = \text{Join}(D_1,\text{DOWN})$

Note: $A$, the result relation, is null initially.

**HN**

Set $W_0 = \text{Project(Select(UP))}$
For $i=1,2,3,4,5$ Do
Begin
   $R_i = \text{Project(Join}(W_{i-1},\text{FLAT}))$
   Set $T_i = R_i$
   For $j=1,2,\ldots,i$ Do
   Begin
      $T_{ij} = \text{Project(Join}(T_{ij-1},\text{DOWN}))$
   End
   Set $A = \text{Union}(A,T_{ij})$
Set $W_i = \text{Project}(\text{Join}(W_{i-1}, UP))$

End

Note: $A$, the result relation, is null initially.

MS

Set $MAGIC = M_0 = \text{Project}(\text{Select}(UP))$

For $i = 2, 3, 4, 5$ Do

Begin

$M_i = \text{Diff}(\text{Join}(M_{i-1}, UP), M_{i-1})$

$MAGIC = \text{Union}(MAGIC, M_i)$

End

Set $MUP = \text{Project}((\text{Union}(\text{Join}(MAGIC, UP), \text{Select}(UP))))$

Set $MFLAT = \text{Project}(\text{Join}(MAGIC, FLAT))$

Set $MF_0 = MFLAT$

For $i = 1, 2, 3, 4, 5$ Begin

$MF = \text{Diff}(\text{Project}(\text{Join}(MUP, MF_{i-1}), UP), MF_{i-1})$

$A = \text{Union}(A, MF)$

End

Set $A = \text{Select}(A)$

Note: $A$, the result relation, is null initially.

References

[BMSU 86]. Bancilhon, F., Maier, D., Sagiv, Y. and Ullman, J.
Magic Sets and Other Strange Ways to Implement Logic Programs.

[BR 86]. Bancilhon, F. and Ramakrishnan, R.
An Amateur's Introduction to Recursive Query Processing Strategies.

[GMN 84]. Gallaire, H., Minder, J., and Nicolas, J-M.
Logic and Databases: A Deductive Approach.
ACM Computing Surveys 16(2), June, 1984.

[HH 87]. Han, J. and Henschen, L.J.
Processing Linear Recursive Database Queries by Level and Cycle Merging.

[HL 86]. Han, J. and Lu, H.
Some Performance Results on Recursive Query Processing.
In Int. Conf. on Data Engineering. IEEE, February, 1986.

[HN 84]. Henschen, L.J. and Naqvi, S.
On Compiling Queries in Recursive First-Order Databases.

[YZ 87]. Yu, C.T. and Zhang, W.
Efficient Recursive Query Processing Using Wavefront Methods.