BENCHMARKING MULTI-RULE RECURSION EVALUATION STRATEGIES

Jeffrey F. Naughton

CS-TR-141-88

March 1988

Benchmarking Multi-Rule Recursion Evaluation Strategies

Jeffrey F. Naughton
Princeton University

March 16, 1988

Abstract

This paper presents an empirical comparison of the Semi-Naive, Generalized Magic Sets, Generalized Counting, and Separable query evaluation strategies as applied to queries on multi-rule recursions. I used each of the methods to evaluate queries over randomly generated relations. For each query there is a critical density range. If the base relations are below the density range, Generalized Magic Sets, Generalized Counting, and Separable provide roughly equal performance and are significantly better than Semi-Naive. Above the density range, Generalized Magic Sets degrades to Semi-Naive, Generalized Counting is much worse than Semi-Naive, while Separable is still significantly better than Semi-Naive. This suggests that special purpose algorithms such as Separable can greatly improve the performance of a recursive query processor.

1 Introduction

In recent years a great deal of attention has been devoted to algorithms for evaluating queries on recursively defined relations. Comparing the performance of these algorithms by an inspection of their descriptions is difficult, especially on complex, multi-rule recursive definitions. However, as answering queries on recursively defined relations promises to be critical to the performance of systems with logic-based query languages, it is important to develop an understanding of how the algorithms compare on various types of queries.

This paper presents empirical statistics on the performance of four evaluation algorithms on three queries. The algorithms considered are Semi-Naive [Ban85], Generalized Magic Sets [BR87], Generalized Counting [BR87,SZ86], and Separable [Nau88]. The queries considered are given in Figures 1, 2, and 3. The results obtained show that, on the queries and relations considered here, Separable is much more efficient than Generalized Magic Sets or Generalized Counting.

The evaluation algorithms were chosen as representative of 4 general approaches to recursive query evaluation:

- Semi-Naive constructs the entire recursively defined relation.
- Generalized Magic Sets attempts to use constants in the query to compute only relevant tuples of the recursively defined relation.

- Generalized Counting, in addition to using query constants to compute only relevant tuples, reduces the arity (number of columns) of the recursively defined relation by storing complete information about partial derivations of answer tuples.

- Separable uses query constants to compute only tuples, and reduces the arity of the recursively defined relation, but does so without recording complete information about partial derivations of answer tuples.

As Generalized Counting and Separable do not apply to all recursions, they must be backed up by a more general algorithm such as Generalized Magic Sets or Semi-Naive. Obviously, there is no reason to include such strategies in a system if their performance is no better than that of more general evaluation algorithms; one of the main goals of this research was to determine the relative performance of these specialized strategies and the more general strategies.

In order to compare query evaluation algorithms, one must choose a database with which to experiment. There are at least three logical choices: data from some real application, structured synthetic data with characteristics that match those of the expected real data, and random data. As there are no databases with recursive capabilities in general use today, I could not find "typical" real data, nor was I confident I could accurately predict what structure "real" data will have; hence I used randomly generated relations.

In comparing query evaluation algorithms, one is primarily concerned with the resources of time and space. However, both are highly dependent on the particular implementation of the algorithm in question. As this work aims to compare the algorithms, and not their implementations, I chose the maximum number of tuples generated during the course of answering the query as the performance metric. (This follows Bancilhon and Ramakrishnan [BR86b].) In addition to the number of tuples, I have included some of the query evalution times in my implementation, as those times may be of some interest.

The evaluation algorithms could be compared analytically, by finding for each a function that gives the number of intermediate tuples as a function of certain properties of the relations. The difficulty here lies not in finding such functions (which is straightforward,) but lies instead in finding formulas for the distribution of these properties in random relations. Barring such a mathematical analysis, the next best thing is to compare the algorithms experimentally, that is, by actually comparing their performance on randomly generated relations.

I tested the algorithms by implementing them on a prototype back-end for deductive databases called IRL (Intermediate Relational Language). The actual benchmark runs were done on IRL running on a DEC Titan workstation, an experimental 15 MIP, 128 MByte workstation developed at DEC's Western Research Lab. More details about IRL appear in Section 3.

Let the *density* of a relation to be the ratio of the number of tuples in the relation to the size of the domains for the columns of a relation. Briefly, my experiments show that for each query there is a critical density range. If the density of the relations is lower than the critical range, the relation is *sparse*; otherwise it is *dense*. The critical density ranges for the queries considered here were low.

```
t(X,Y) :- a(X,W) & t(W,Y).
t(X,Y) :- b(X,W) & t(W,Y).
t(X,Y) :- a(X,Y).
t(X,Y) :- b(X,Y).

t(v,Y)?
```

Figure 1: Program and query for query A.

I found that if the base relations are sparse, Generalized Magic Sets, Generalized Counting, and Separable provided roughly equal performance and were all superior to Semi-Naive. If the base relations are not sparse, Generalized Magic Sets degenerates to Semi-Naive, Generalized Counting is significantly worse than Semi-Naive, while Separable is still significantly superior to Semi-Naive.

To my knowledge this is the first work comparing these evaluation algorithms on multiple recursive rule recursions. In perhaps the most closely related work, Bancilhon and Ramakrishnan [BR86b] provide an analytic comparison of a number of evaluation methods over four single recursive rule queries on some highly structured databases. Their results showed that Counting was better than Magic, while Magic was better than Semi-Naive. The discrepancy between their results and my results arises for two reasons: first, Bancilhon and Ramakrishnan considered only single recursive rule queries; second, their sample databases were highly structured.

In Section 2 I describe the queries and sample databases used in the benchmarks. Section 3 describes how the algorithms were implemented and tested in our prototype system. Section 4 presents the results of the experiments. Section 5 discusses these results.

Appendix A describes the algorithm Separable, and is included here to make this paper self-contained, as the primary description of Separable [Nau88] is not yet widely available. Appendix B lists the IRL code to implement the queries considered here. Finally, Appendix C includes graphs of the data tables included in the text.

2 Queries and Data

In this work I considered three different queries. The queries, expressed in Datalog, appear in Figures 1, 2, and 3. In each of the queries the relations `a`, `b`, `c`, and `d` are *base relations*, that is, they are defined by sets of tuples explicitly stored in the database. The relation `t` is defined instead implicitly by the rules in the programs.

The symbol "`:-`" can be read as "if." Uppercase letters represent variables, while lowercase represent constants. Thus the first line in Figure 1 says "the tuple `(X,Y)` is in `t` if, for some `W`, the tuple `(X,W)` is in `a` and the tuple `(W,Y)` is in `t`. The query `t(v,Y)?` asks for all values of `Y` that appear in the second attribute of a tuple of `t` in which `v` appears in the first column. (This query is equivalent to the relational algebra expression $\sigma_{1='v'}(t)$.)

The program in Query A defines `t` to be the transitive closure of the union of the base

```
t(X,Y) :- a(X,W) & t(W,Y).
t(X,Y) :- t(X,W) & b(W,Y).
t(X,Y) :- a(X,Y).
t(X,Y) :- b(X,Y).

t(v,Y)?
```

Figure 2: Program and query for query B.

```
t(X,Y,Z) :- a(X,U) & t(U,Y,Z).
t(X,Y,Z) :- b(Y,W) & t(X,W,Z).
t(X,Y,Z) :- c(Z,V) & t(X,Y,V).
t(X,Y,Z) :- d(X,Y,Z).

t(v,Y,Z)?
```

Figure 3: Program and query for query C.

relations `a` and `b`, while the program in Query B defines `t` to be the join of the transitive closures of relations `a` and `b`. Both queries A and B are representative of recursive procedures in which there are two options for how to recurse. Query C is included as an example of a non-binary program.

Again, as my goal was to benchmark the algorithms and not IRL, I used relations in which all fields contain 32-bit integers. (Using strings and so forth would affect the cost of doing comparisons and storing the tuples, but would not affect the number of tuples created by the methods in answering the queries.)

The important parameters for a relation are its number of tuples, and its "density," that is, the ratio of the number of tuples in the relation to the size of the domains for the relation's columns. To simplify matters, I considered only relations in which the domains for all columns were identical. Then to generate a k-ary relation with n_t tuples and n_d elements in its domain, I repeatedly generated random k tuples of integers in the range 0 to n_d, until n_t distinct tuples were produced.

One modification was necessary when testing the Generalized Counting algorithm. As Generalized Counting fails to terminate when the data is cyclic, I needed to generate "random" acyclic relations. I did this by rejecting all tuples in which the second attribute was less than or equal to the first. (Generalized Counting only applied to Query A, which has only binary relations.)

3 Implementation

As noted in the Introduction, Semi-Naive is described in [Ban85], while Generalized Magic Sets is described in [BR87] and Generalized Counting is described in [BR87] and [SZ86]. Counting and Magic methods have received a lot of attention; for more information about those methods one may consult, in addition to the references above, [BMSU86,SZ87], and the survey paper [BR86a]. Due to space limitations I will not further describe these algorithms here. The IRL code to implement these methods on the queries considered here appears in Appendix B.

With the exception of Generalized Counting, all the evaluation algorithms apply to all the queries. On Queries B and C, Generalized Counting will not terminate (unless the base relations are empty), so data for Generalized Counting is only presented for Query A.

IRL (Intermediate Relational Language) is intended to be a flexible, high-performance back end for databases with logic-based query languages. Its specification (but not implementation) is similar to the NAIL! ICODE [Ull86]. IRL has three types of operations: data manipulation, I/O, and control.

The IRL data manipulation operations implement an extended relational algebra. The IRL I/O instructions are provided to initialize base relations from UNIX files, and to print results during the execution of the program. The flow control instructions consist of conditional and unconditional goto's.

IRL is implemented in C. It has no facilities for concurrency control or crash recovery. However, as IRL is fast and flexible, it provides an ideal testbed for query evaluation algorithms.

4 Results

The benchmarks consist of a series of queries at varying densities and relation sizes. At each density and relation size, each method was used to evaluate 100 queries. The 100 queries were divided into 10 queries on each of 10 randomly generated relations; the query constants used were the numbers 0 through 9. The maximum number of intermediate tuples generated during each query evaluation were averaged over the 100 trials. Graphs for these tables appear in Appendix B.

Table 1 lists the results of evaluating Query A over randomly generated relations of varying density, where each base relation has 200 tuples. A graph of Table 1 can be found in Appendix B. Note that, for a given number of tuples, as the density of a relation increases, the number of tuples in the recursively defined relation first increases as the database becomes more and more "connected." However, increasing the density while keeping the number of tuples constant corresponds to decreasing the size of the domain of the relation, thus reducing the maximum possible size for the recursively defined relation. This is reflected in the decrease in the number of tuples produced by Semi-Naive from density 0.9 to density 1.0.

While the time to evaluate a query depends on the implementation as much as the evaluation algorithm, time is undoubtably of some interest. Hence Table 2 lists the average

density	snaive	magic	sep
0.1	514	1	1
0.2	690	2	2
0.3	1172	9	3
0.4	1818	29	6
0.5	2089	426	17
0.6	6909	1053	31
0.7	10614	6812	80
0.8	22312	11916	105
0.9	26947	13651	116
1.0	25566	16682	129

Table 1: Generated tuples vs. density, Query A, $n = 200$

density	snaive	magic	sep
0.1	0.21	0.03	0.02
0.2	0.31	0.03	0.02
0.3	0.65	0.04	0.02
0.4	1.39	0.07	0.02
0.5	8.39	0.58	0.04
0.6	13.94	1.36	0.05
0.7	40.85	10.77	0.09
0.8	57.33	23.78	0.10
0.9	56.74	29.08	0.10
1.0	60.23	38.92	0.12

Table 2: Seconds vs. density, Query A, $n = 200$

base tuples	snaive	magic	sep
200	908	3.81	2.38
400	1639	2.32	2.15
600	2431	3.24	2.32
800	3204	1.75	1.75
1000	4054	2.84	2.38
1200	4879	3.66	2.27
1400	5579	2.49	2.29
1600	6344	1.93	1.93
1800	7148	1.89	1.89
2000	7945	1.75	1.75

Table 3: Intermediate tuples vs. base tuples, Query A, $d = 0.25$.

base tuples	snaive	magic	sep
20	271	182	13
40	993	624	24
60	2177	1366	36
80	3688	2064	44
100	5939	3620	59
120	8827	5488	73
140	11293	7116	87
160	15883	10540	104
180	19249	11656	107
200	24966	16682	129

Table 4: Base vs. generated tuples, Query A, $d = 1.00$.

times (sum of system and user) to answer Query A. These tests were run on a lightly loaded titan, but there were other users active during the course of the benchmarks.

Table 3 gives the average performance of Semi-Naive, Generalized Magic Sets, and Separable on random relations with densities of .25 and sizes varying from 200 to 2000 tuples.

Table 4 shows the performance of Semi-Naive, Generalized Magic Sets, and Separable on random relations of density 1.0, ranging in size from 20 to 200 tuples.

Table 5 compares Semi-Naive, Generalized Magic Sets, Separable, and Counting on acyclic random relations of varying density, all with 100 tuples. The effect of the smaller domain size reducing the possible number of answer tuples is clear from density 1.5 on.

At low densities, Counting and Separable are roughly equivalent, while Generalized Magic Sets is about a factor of two worse. (See Table 6.)

density	snaive	counting	magic	sep
0.5	354	9	13	7
1.0	573	54	100	23
1.5	610	303	211	31
2.0	593	1076	282	32
2.5	481	1939	250	27
3.0	420	4250	265	26
3.5	316	6747	182	21
4.0	268	9561	161	19

Table 5: Generated tuples vs. density, Query A, acyclic relations, $n = 100$.

base tuples	snaive	counting	magic	sep
200	912	7.7	15.0	7.6
400	1774	8.2	17.2	8.0
600	2498	6.8	12.2	6.8
800	3468	6.9	12.7	6.8
1000	4280	6.6	11.9	6.6
1200	5145	7.1	13.8	7.1
1400	6040	7.5	14.2	7.5
1600	6836	6.7	12.5	6.7
1800	7688	8.2	17.0	8.2
2000	8679	6.8	12.6	6.8

Table 6: Base vs. generated tuples, Query A, acyclic relations, $d = 0.5$.

base tuples	snaive	counting	magic	sep
20	10	18	4	3
40	45	482	29	8
60	102	1720	74	13
80	171	3750	124	17
100	271	7603	205	22

Table 7: Base vs. generated tuples, Query A, acyclic relations, $d = 4.0$.

density	snaive	magic	sep
0.2	579	1	2
0.4	942	5	4
0.6	1608	8	5
0.8	4427	218	20
1.0	8431	596	45
1.2	10344	1536	67
1.4	11657	3733	92
1.6	11501	4172	96
1.8	10573	4712	87
2.0	8711	5320	99

Table 8: Generated tuples vs. density, Query B, $n = 200$.

density	snaive	magic	sep
0.2	496	2	2
0.4	555	7	4
0.6	1253	22	14
0.8	3572	314	61
1.0	8637	1699	165
1.2	11271	2284	252
1.4	16206	5802	423
1.6	16598	9365	560
1.8	14958	9218	547
2.0	12092	7864	462

Table 9: Density vs. generated tuples, Query C, $n = 50$.

Table 7 shows the relative performances at higher density with acyclic data.

Table 8 compares Semi-Naive, Generalized Magic Sets, and Generalized Counting on relations of varying density, each with 200 tuples. Again, at higher densities Generalized Magic Sets approaches Semi-Naive, although this density is higher than that in Query A. This is because Query A effectively takes the union of the relations `a` and `b`, whereas Query B does not. Comparing Semi-Naive, Generalized Magic Sets, and Separable at high and low densities for varying relation sizes yielded similar results for Query B as for Query A (acyclic data), so the results of those trials are omitted.

Finally, Table 9 compares Semi-Naive, Generalized Magic Sets, and Separable on query C for varying densities of random relations of size 50. Again, comparing Semi-Naive, Generalized Magic Sets, and Separable at high and low densities for varying relation sizes yielded similar results for Query C as for Query A, so the results of those trials are omitted.

5 Conclusion

The data presented here demonstrates that the performance of these evaluation algorithms depends critically on the density of the base relations. If the base relations are sparse, any of the algorithms (other than Semi-Naive) provides acceptable performance. However, if the relations are dense Separable is preferable. This can be explained as follows.

Generalized Magic Sets computes the portion of the recursively defined relation deemed relevant by the "magic set." As the relation becomes more dense, the "magic set" comprises more and more of the database. Once the magic set includes the entire database, Generalized Magic Sets and Semi-Naive compute exactly the same relation.

Generalized Counting reduces the "arity" of the recursively defined relation, which reduces the number of tuples computed. However, it stores complete information about every possible derivation of an answer tuple. (Here, a derivation is just the list of rule applications that produce the answer.) As the relation becomes more dense, there are an exponential number of ways to derive answers, and Generalized Counting computes far more information than Semi-Naive.

Separable reduces the "arity" of the recursively defined relation while storing only minimal information about derivations. (It records that an intermediate tuple has been produced, without recording the sequence of rule applications by which it was produced.)

The density at which Separable begins to dominate the other algorithms is low. For example, Separable out performs Generalized Magic Sets by a factor of 110 on relations with density 0.8 and 200 tuples. To put this into perspective, consider the select-project-join query $Q = \sigma_{\$1=c}(a \bowtie b)$, where each of a and b have densities 0.8 and have 200 tuples. This means that the domains for the columns of a and b are of size 250. First, the join $a \bowtie b$ can be expected to yield only $200 * 200/250 = 160$ tuples. If we assume the constant c is randomly chosen among the 250 domain values, c can be expected to appear in the first column of the result $160/250 = 0.64$ times, hence the answer to Q has an expected size of 0.64 tuples. The point is that these relations are so sparse that evaluating select-project-join queries over them is trivial.

Furthermore, the density range over which the algorithms go from being roughly equivalent to being very different is small. This mirrors the situation with the size of connected components in random graphs. For example, asymptotically a random (undirected) graph of density less than $n/2$ has a largest connected component of size $O(\log n)$, whereas a random graph of density equal to $n/2$ has a largest connected component of size $O(n^{2/3})$ [Bol85]. Of course, as relations are directed hypergraphs rather than undirected graphs, and the performance of the evaluation algorithm depends on the shape as well as the size of the components of these hypergraphs, the analogy is not precise.

As the Separable algorithm does not apply to all recursions, it must be supplemented by other evaluation algorithms. However, given the performance of Generalized Magic Sets and Generalized Counting on even the small relations considered here, special purpose algorithms such as Separable will significantly improve system performance on recursive queries.

References

[Ban85] Francois Bancilhon. Naive evaluation of recursively defined relations. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems — Integrating Database and AI Systems*, Springer-Verlag, 1985.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15, 1986.

[Bol85] Béla Bollobás. *Random Graphs.* Academic Press, London, 1985.

[BR86a] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1986.

[BR86b] Francois Bancilhon and Raghu Ramakrishnan. Performance evaluation of data intensive logic programs. In *Preprints of Workshop on Foundations of Deductive Databases and Logic Programming*, August 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269–283, 1987.

[Nau88] Jeffrey F. Naughton. Compiling separable recursions. In *Proceedings of the SIGMOD International Symposium on Management of Data*, 1988.

[SZ86] Domenico Sacca and Carlo Zaniolo. The generalized counting methods for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, 1986.

[SZ87] Domenico Sacca and Carlo Zaniolo. Magic counting methods. In *Proceedings of the ACM-SIGMOD Symposium on the Management of Data*, pages 49–59, 1987.

[Ull86] Jeffrey D. Ullman. NAIL! ICODE summary: C/SQL version. 1986. Unpublished memorandum, Stanford University.

A The Separable Algorithm

In this appendix we present the separable algorithm, which evaluates queries of the form "column = constant" on relations defined by separable recursions. First, however, we must define what constitutes a "separable" recursion. We begin with some auxiliary definitions.

Definition A.1 A predicate instance p_1 is *connected* to a predicate instance p_2 if p_1 shares a variable with p_2, or shares a variable with a predicate instance connected to p_2.

Definition A.2 A subset of predicate instances C in a Datalog rule is a *connected set* if

1. For every pair of predicate instances p_1 and p_2 in C, p_1 and p_2 are connected, and
2. No predicate instance in C shares a variable with any predicate instance not in C.

Definition A.3 Let r be a linear recursive rule and let t be the recursive predicate in r. Then r contains *shifting variables* if there is some variable X such that X appears in position t^i in the head of r and in position t^j in the body of r, where $t^i \neq t^j$.

Definition A.4 (Separable Recursions) Let t be defined by n recursive rules r_1 through r_n. Furthermore, let t_i^h be the argument positions of t such that in the instance of t at the head of rule r_i, each argument position in t_i^h shares a variable with a nonrecursive predicate in the body of r_i. Similarly, let t_i^b be the argument positions of t such that in the instance of t in the body of rule r_i, each argument position in t_i^b shares a variable with a nonrecursive predicate in the body of r_i. Then the definition of t is a *separable recursion* if

1. For $1 \leq i \leq n$, r_i has no shifting variables, and
2. For $1 \leq i \leq n$, $t_i^h = t_i^b$, and
3. For $1 \leq i \leq n$ and $i < j \leq n$, either $t_i^h = t_j^h$ or t_i^h and t_j^h are disjoint, and
4. For $1 \leq i \leq n$, removing the instance of t from the body of r_i leaves a connected set.

Figure 4 gives a general schema for evaluating a selection on a separable recursion. The $carry_i$, the $seen_i$, and ans are relation-valued variables. The f_i, g_i, and h are relational operators. In addition to the arguments listed, the f_i, g_i, and h may involve relations and constants appearing in the rules and in the query.

The details of how this schema is instantiated are given in [Nau88].

B Sample IRL Code

Here we present the resulting IRL code that implements the evaluation methods on the queries considered in this paper.

1) init $carry_1$;
2) $seen_1 := carry_1$;
3) while $carry_1$ not empty do
4) $\quad carry_1 := f_1(carry_1)$;
5) $\quad seen_1 := seen_1 \cup carry_1$;
6) $\quad carry_1 := carry_1 - seen_1$;
7) endwhile;
8) $carry_2 := g_2(seen_1)$;
9) $seen_2 := carry_2$;
10) while $carry_2$ not empty do
11) $\quad carry_2 := f_2(carry_2)$;
12) $\quad seen_2 := seen_2 \cup carry_2$;
13) $\quad carry_2 := carry_2 - seen_2$;
14) endwhile;
15) $ans := h(seen_1, seen_2)$;

Figure 4: A schema for evaluating single selections on separable recursions.

```
                readin(a, a);
                readin(b, b);

                tmp(X,Y) := a(X,Y) + b(X,Y);
                t(X,Y) := tmp(X,Y);

#ansLoop:       ifempty tmp(X,Y) goto #done;
                ttmp1(X,Y) := a(X,W) & tmp(W,Y);
                ttmp2(X,Y) := b(X,W) & tmp(W,Y);
                tmp(X,Y)   := ttmp1(X,Y) + ttmp2(X,Y);
                tmp(X,Y)   := tmp(X,Y) - t(X,Y);
                t(X,Y)     := t(X,Y) + tmp(X,Y);
                goto #ansLoop;

#done:          ans(Y) := seed(X) & t(X,Y);
.
```

Figure 5: IRL code for Semi-Naive on Query A

```
#start:       readin(seed, seed);
              readin(a, a);
              readin(b, b);

              magtmp(X) := seed(X);
              magic(X) := magtmp(X);
#magloop:     ifempty magtmp(X) goto #initAns;
              magatmp(W) := magtmp(X) & a(X,W);
              magbtmp(W) := magtmp(X) & b(X,W);
              magtmp(X) := magatmp(X) + magbtmp(X);
              magtmp(X) := magtmp(X) - magic(X);
              magic(X) := magic(X) + magtmp(X);
              goto #magloop;

#initAns:     ttmp1(X,Y) := magic(X) & a(X,Y);
              ttmp2(X,Y) := magic(X) & b(X,Y);
              tmp(X,Y) := ttmp1(X,Y) + ttmp2(X,Y);
              t(X,Y) := tmp(X,Y);

#ansLoop:     ifempty tmp(X,Y) goto #done;
              ttmp1(X,Y) := magic(X) & a(X,W) & tmp(W,Y);
              ttmp2(X,Y) := magic(X) & b(X,W) & tmp(W,Y);
              tmp(X,Y)   := ttmp1(X,Y) + ttmp2(X,Y);
              tmp(X,Y)   := tmp(X,Y) - t(X,Y);
              t(X,Y)     := t(X,Y) + tmp(X,Y);
              goto #ansLoop;

#done:        ans(Y) := seed(X) & t(X,Y);
.
```

Figure 6: IRL code for GMS on Query A

```
#start:         readin(seed, seed);
                readin(a, a);
                readin(b, b);

                cnttmp(K,X) := seed(X) & cond(K=1);
                cnt(K,X) := cnttmp(K,X);
#cntloop:       ifempty cnttmp(K,X) goto #initAns;
                cntatmp(K1,W) := cnttmp(K,X) & a(X,W) & K1 = 2*K;
                cntbtmp(K1,W) := cnttmp(K,X)
                                 & b(X,W)
                                 & K1 = 2*K+1;
                cnttmp(K,X) := cntatmp(K,X) + cntbtmp(K,X);
                cnttmp(K,X) := cnttmp(K,X) - cnt(K,X);
                cnt(K,X) := cnt(K,X) + cnttmp(K,X);
                goto #cntloop;

#initAns:       ttmp(K,Y) := cnt(K,X) & c(X,Y);
                t(K,Y) := ttmp(K,Y);
#ansLoop:       ifempty ttmp(K,Y) goto #done;
                ttmpa(K1,Y) := ttmp(K,Y)
                               & K1 = K/2
                               & K/2*2 = K;
                ttmpb(K1,Y) := ttmp(K,Y)
                               & K1=(K-1)/2
                               & ((K-1)/2)*2=K-1;
                ttmp(K,Y) := ttmpa(K,Y) + ttmpb(K,Y);
                ttmp(K,Y) := ttmp(K,Y) - t(K,Y);
                t(K,Y)   := t(K,Y) + ttmp(K,Y);
                goto #ansLoop;

#done:          ans(X) := t(I,X) & I = 0;
.
```

Figure 7: IRL code for GC on Query A

```
#start:         readin(seed, seed);
                readin(a, a);
                readin(b, b);

                tmp(X) := seed(X);
                seen(X) := tmp(X);
#loop:          ifempty tmp(X) goto #done;
                catmp(W) := tmp(X) & a(X,W);
                cbtmp(W) := tmp(X) & b(X,W);
                tmp(X) := catmp(X) + cbtmp(X);
                tmp(X) := tmp(X) - seen(X);
                seen(X) := seen(X) + tmp(X);
                goto #loop;

#done:          ansa(X) := seen(W) & a(W,X);
                ansb(X) := seen(W) & b(W,X);
                ans(X) := ansa(X) + ansb(X);
.
```

Figure 8: IRL code for Sep on Query A

```
                readin(a, a);
                readin(b, b);

                tmp(X,Y) := a(X,Y) + b(X,Y);
                t(X,Y) := tmp(X,Y);

#ansLoop:       ifempty tmp(X,Y) goto #done;
                ttmp1(X,Y) := a(X,W) & tmp(W,Y);
                ttmp2(X,Y) := tmp(X,W) & b(W,Y);
                tmp(X,Y)   := ttmp1(X,Y) + ttmp2(X,Y);
                tmp(X,Y)   := tmp(X,Y) - t(X,Y);
                t(X,Y)     := t(X,Y) + tmp(X,Y);
                goto #ansLoop;

#done:          ans(Y) := seed(X) & t(X,Y);
.
```

Figure 9: IRL code for Semi-Naive on Query B

```
#start:      readin(seed, seed);
             readin(a, a);
             readin(b, b);

             magtmp(X) := seed(X);
             magic(X) := magtmp(X);
#magloop:    ifempty magtmp(X) goto #initAns;
             magtmp(W) := magtmp(X) & a(X,W);
             magtmp(X) := magtmp(X) - magic(X);
             magic(X) := magic(X) + magtmp(X);
             goto #magloop;

#initAns:    tmp(X,Y) := magic(X) & a(X,Y);
             t(X,Y) :=   magic(X) & b(X,Y);
             t(X,Y) := tmp(X,Y) + t(X,Y);
             tmp(X,Y) := t(X,Y);

#ansLoop:    ifempty tmp(X,Y) goto #done;
             ttmp1(X,Y) := magic(X) & a(X,W) & tmp(W,Y);
             ttmp2(X,Y) := magic(X) & tmp(X,W) & b(W,Y);
             tmp(X,Y)   := ttmp1(X,Y) + ttmp2(X,Y);
             tmp(X,Y)   := tmp(X,Y) - t(X,Y);
             t(X,Y)     := t(X,Y) + tmp(X,Y);
             goto #ansLoop;

#done:       ans(Y) := seed(X) & t(X,Y);
.
```

Figure 10: IRL code for GMS on Query B

```
#start:         readin(seed, seed);
                readin(a, a);
                readin(b, b);

                tmp(X) := seed(X);
                seena(X) := tmp(X);
#aloop:         ifempty tmp(X) goto #initb;
                tmp(W) := tmp(X) & a(X,W);
                tmp(X) := tmp(X) - seena(X);
                seena(X) := seena(X) + tmp(X);
                goto #aloop;

#initb:         tmp(X) := seena(X);
                seenb(X) := tmp(X);

#bloop:         ifempty tmp(X) goto #done;
                tmp(Y) := tmp(Z) & b(Z,Y);
                tmp(Y) := tmp(Y) - seenb(Y);
                seenb(Y) := seenb(Y) + tmp(Y);
                goto #bloop;

#done:          ans(Y) := seenb(Y);
.
```

Figure 11: IRL code for Sep on Query B

```
                tmp(X,Y,Z) := d(X,Y,Z);
                t(X,Y,Z) := tmp(X,Y,Z);

#ansLoop:       ifempty tmp(X,Y,Z) goto #done;
                ttmp1(X,Y,Z) := a(X,W) & tmp(W,Y,Z);
                ttmp2(X,Y,Z) := tmp(X,W,Z) & b(W,Y);
                ttmp3(X,Y,Z) := tmp(X,Y,W) & c(W,Z);
                tmp(X,Y,Z)   := ttmp1(X,Y,Z) + ttmp2(X,Y,Z);
                tmp(X,Y,Z) := tmp(X,Y,Z) + ttmp3(X,Y,Z);
                tmp(X,Y,Z)   := tmp(X,Y,Z) - t(X,Y,Z);
                t(X,Y,Z)     := t(X,Y,Z) + tmp(X,Y,Z);
                goto #ansLoop;

#done:          ans(Y,Z) := t(X,Y,Z) & seed(X);
.
```

Figure 12: IRL code for Semi-Naive on Query C

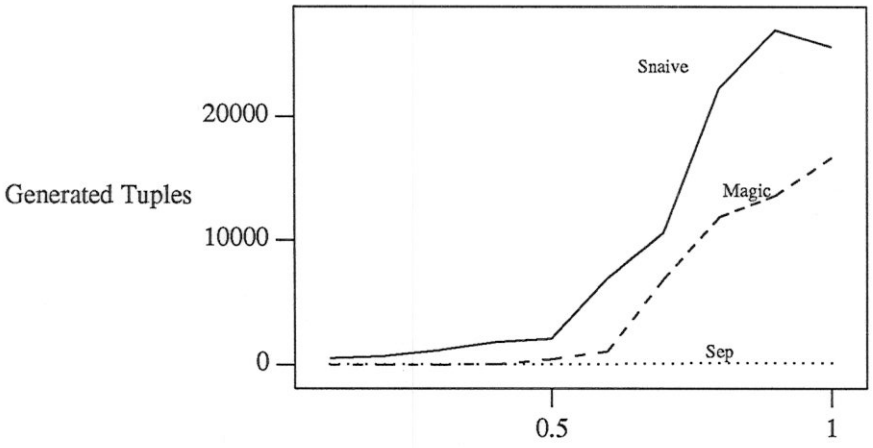
```
#start:        readin(seed, seed);
               readin(a, a);
               readin(b, b);
               readin(d, d);

               magtmp(X) := seed(X);
               magic(X) := magtmp(X);
#magloop:      ifempty magtmp(X) goto #initAns;
               magtmp(W) := magtmp(X) & a(X,W);
               magtmp(X) := magtmp(X) - magic(X);
               magic(X) := magic(X) + magtmp(X);
               goto #magloop;

#initAns:      tmp(X,Y,Z) := magic(X) & d(X,Y,Z);
               t(X,Y,Z) := tmp(X,Y,Z);

#ansLoop:      ifempty tmp(X,Y,Z) goto #done;
               ttmp1(X,Y,Z) := magic(X) & a(X,W) & tmp(W,Y,Z);
               ttmp2(X,Y,Z) := magic(X) & tmp(X,W,Z) & b(W,Y);
               ttmp3(X,Y,Z) := magic(X) & tmp(X,Y,W) & c(W,Z);
               tmp(X,Y,Z)   := ttmp1(X,Y,Z) + ttmp2(X,Y,Z);
               tmp(X,Y,Z) := tmp(X,Y,Z) + ttmp3(X,Y,Z);
               tmp(X,Y,Z)   := tmp(X,Y,Z) - t(X,Y,Z);
               t(X,Y,Z)     := t(X,Y,Z) + tmp(X,Y,Z);
               goto #ansLoop;

#done:         ans(Y,Z) := seed(X) & t(X,Y,Z);
.
```

Figure 13: IRL code for GMS on Query C

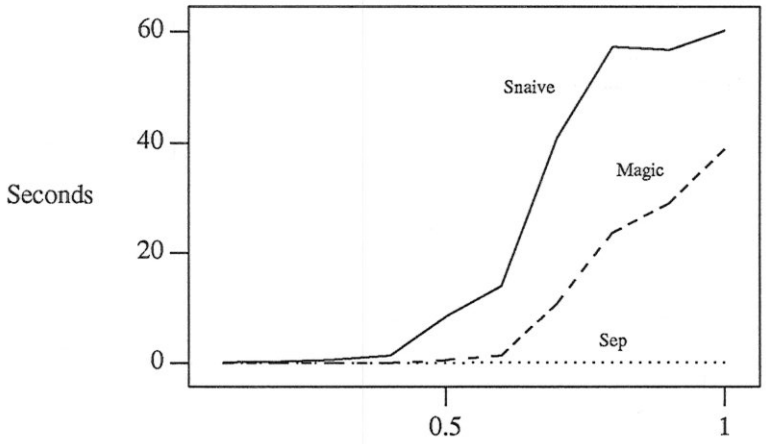
```
#start:     readin(seed, seed);
            readin(a, a);
            readin(b, b);
            readin(d, d);

            tmpa(X) := seed(X);
            seena(X) := tmpa(X);
#loop:      ifempty tmpa(X) goto #initbc;
            tmpa(W) := tmpa(X) & a(X,W);
            tmpa(X) := tmpa(X) - seena(X);
            seena(X) := seena(X) + tmpa(X);
            goto #loop;

#initbc:    tmpbc(Y,Z) := seena(X) & d(X,Y,Z);
            seenbc(Y,Z) := tmpbc(Y,Z);

#bloop:     ifempty tmpbc(Y,Z) goto #done;
            tmpbc1(Y,Z) := tmpbc(W,Z) & b(W,Y);
            tmpbc2(Y,Z) := tmpbc(Y,W) & c(W,Z);
            tmpbc(Y,Z) := tmpbc1(Y,Z) + tmpbc2(Y,Z);
            tmpbc(Y,Z) := tmpbc(Y,Z) - seenbc(Y,Z);
            seenbc(Y,Z) := seenbc(Y,Z) + tmpbc(Y,Z);
            goto #bloop;

#done:      ans(Y,Z) := seenbc(Y,Z);
.
```

Figure 14: IRL code for Sep on Query C

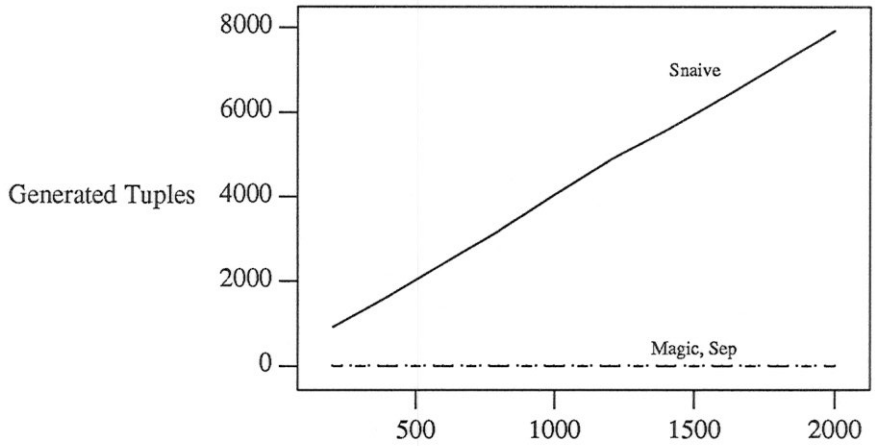
Appendix C.



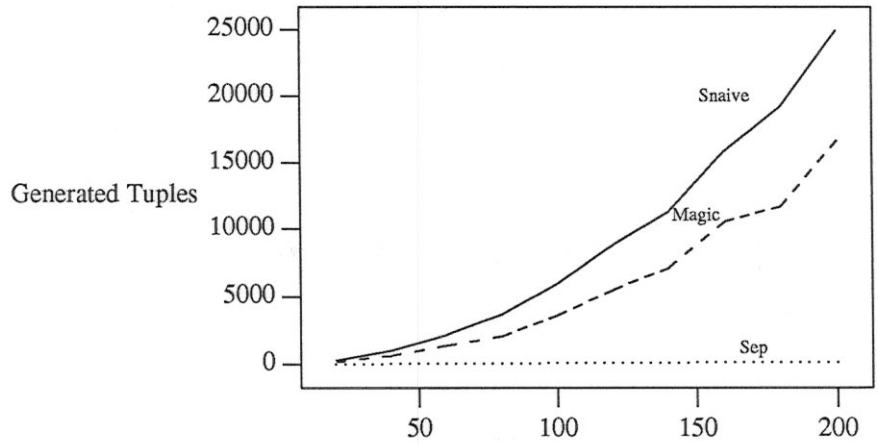
Density, Query A, n = 200.
Graph for Table 1.



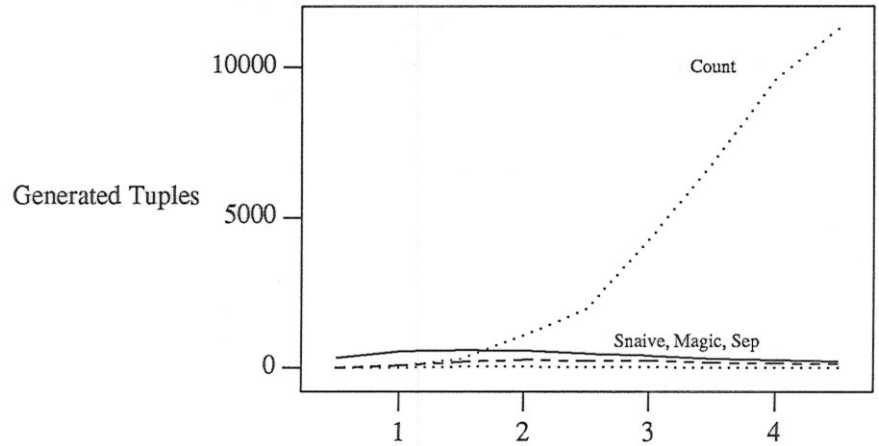
Density, Query A, n = 200.
Graph for Table 2.



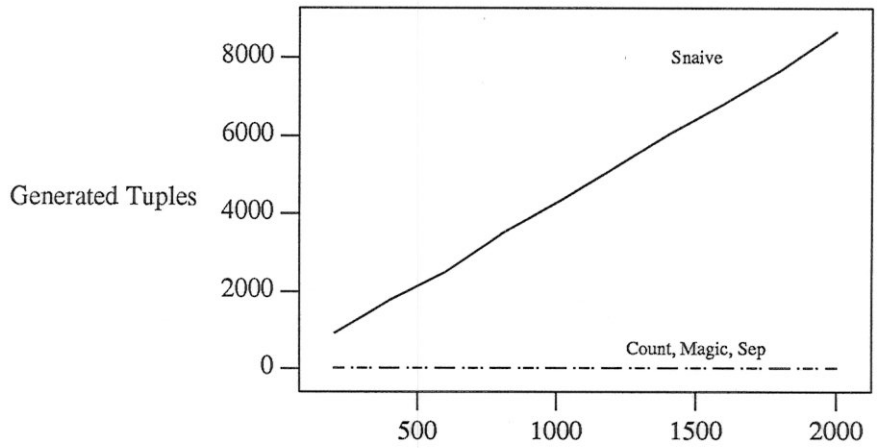
Base Tuples, Query A, d = 0.25.
Graph for Table 3.



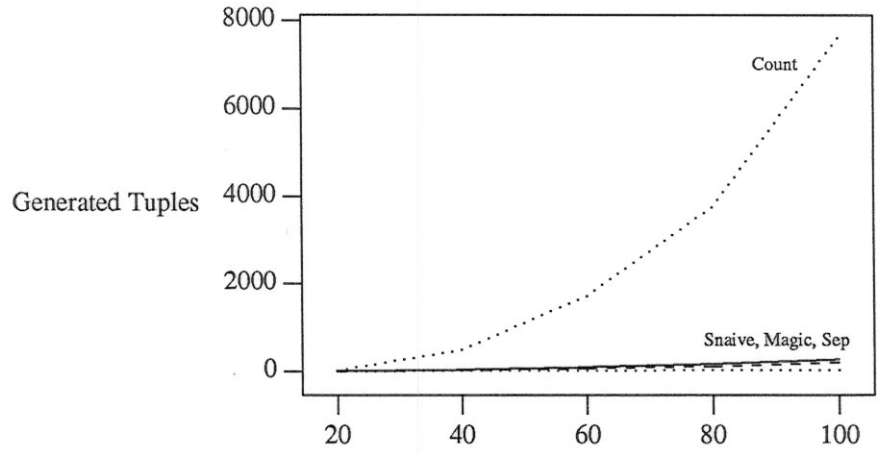
Base Tuples, Query A, d = 1.00
Graph for Table 4.



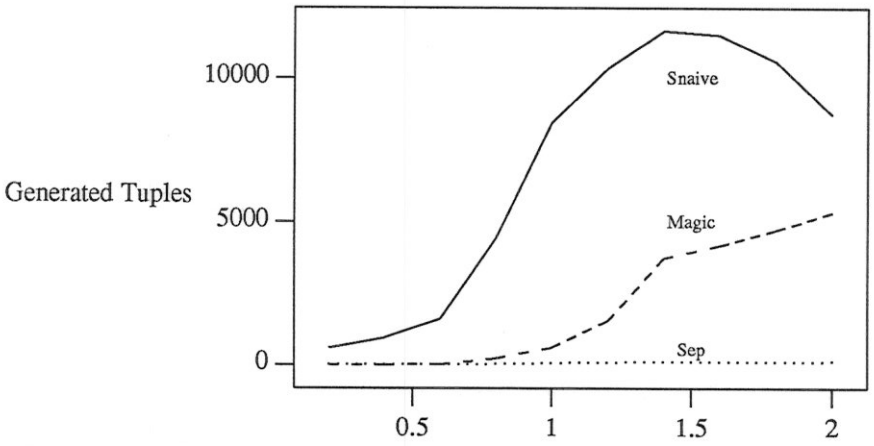
Density, Query A, acyclic relations, n = 100
Graph for Table 5.



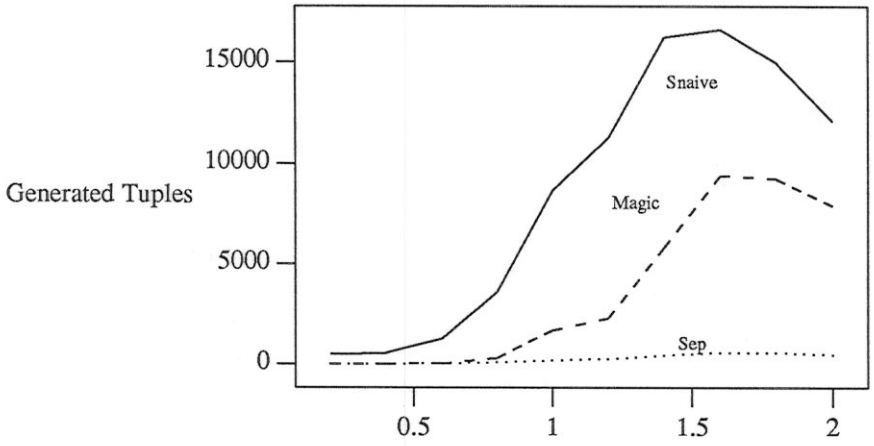
Base tuples, Query A, acyclic relations, d = 0.5
Graph for Table 6.



Base tuples, Query A, acyclic relations, d = 4.0
Graph for Table 7.



Density, Query B, n = 200
Graph for Table 8.



Density, Query C, n = 50
Graph for Table 9.