A Synthetic Join Benchmark for Evaluating DBMS/KBS Hardware and Software

A. J. Marsh
and
S. H. Lavington

Internal Report CSM-173, July 1992

Department of Computer Science
University of Essex
Colchester
CO4 3SQ
UK

Tel: 0206 872 677
e-mail: lavington@uk.ac.essex

A Synthetic Join Benchmark for Evaluating DBMS/KBS Hardware and Software

A. J. Marsh
and
S. H. Lavington

Dept of Computer Science, University of Essex, UK.

1. Introduction

The relational *join* operation has long been used as a guide to the performance of database management systems (DBMS). This is because:

(a) *join* operations are important in real commercial database systems;

(b) the *join* operation, being diadic, is more difficult to optimise - especially in systems that distribute relations amongst several storage 'nodes'.

The relational *join* is a more general form of classical set intersection. In AI and knowledge-base systems (KBS) set intersection is recognised, along with pattern-directed search and transitive closure, as a key generic operation. Furthermore, AI paradigms such as constraint satisfaction and production systems make explicit use of join-like operations as part of their main computational activity. Thus, the *join* operation is a good test of the performance of knowledge-manipulation systems.

There is a need for an easy-to-use *join* test that will allow rapid evaluation of the capabilities of DBMS/KBS platforms. The test is seen as just one of several analytical tools available to experimenters. The requirements are for a benchmark that is:

(i) portable and easily-understood;

(ii) well-defined, so that it can be implemented in several source languages;

(iii) independent of complex data-formats or user-level DBMS facilities, so that it can be used on experimental/prototype systems.

By way of further explanation for requirements (ii) and (iii), the AI community normally uses languages such as PROLOG or LISP for implementing knowledge-based systems. It is well-known that such languages often prove inefficient when handling large volumes of data. Nevertheless, the end-user desire for 'smarter' information systems does imply a synthesis of DBMS and AI insights. A benchmark that allows comparison of the data-manipulation capabilities of languages as diverse as SQL, C, LISP, and PROLOG is obviously useful in facilitating contact between advanced KBS researchers and the real world.

2. The benchmark

The data for the benchmark is generated synthetically, so as to give control over the cardinality of the relations to be joined, the range of field-values (attribute-values) being compared, and the cardinality of the resulting output relation. For simplicity, the two input relations have the same cardinality, n. This n, which is a readily-understood indicator of total volume of data, can conveniently be the control variable in a series of *join* tests. In this way the scalability, or extensibility, of various parallel platforms can be revealed.

The data-values (i.e. field, or attribute values) are chosen so that the output relation has 10 percent of the cardinality of the two input relations. The choice of 10 percent is somewhat arbitrary, but is felt to represent typical activity in practical commercial systems.

The data-values for the synthetic benchmark are all 32-bit integers. Although this is a gross simplification of the (possibly variable-length) character strings that often occur in commercial DBMS, the use of integers does allow us: (a) to generate the data easily; (b) to analyse the performance of hashing/indexing schemes. The benchmark uses simple three-field relations. Once again, this format is employed so as to make data-generation and result-analysis straight-forward. Since some software systems only use the first field ('argument') for indexing, the *join* test is actually performed in variants with a different pair of *join*-fields being specified for each variant. The first variant (Test(a)) permits the simplest form of indexing to detect any join-matches; the second variant (Test(b)) is a more revealing test of indexing capability.

More formally, let the two input relations be R, S. These are each of arity three and of equal cardinality, n. Data values of relation R, field 1, range from 1 to n, with each value only appearing once; similarly for fields 2 & 3. Values for all three fields are placed randomly.

Data values of relation S, field 1, range from $(n - \lfloor n/10 \rfloor + 1)$ to $(2n - \lfloor n/10 \rfloor)$ with each value only appearing once. The data values for fields 2 and 3 range from 1 to n with each value only appearing once, in a manner similar to that for relation R.

For various values of the control variable n, two tests are performed:

Test (a): $R \bowtie_{3=1} S$ Test (b): $S \bowtie_{1=3} R$

The join activity may be represented diagrammatically as in Figure 1. Note that the data in Figure 1 has been simplified for ease of visualisation, by placing data values for field 3 of relation R in ascending order. As may be seen from Figure 1, the resulting output relations for the 10 percent equi-join tests can easily be checked for correctness.

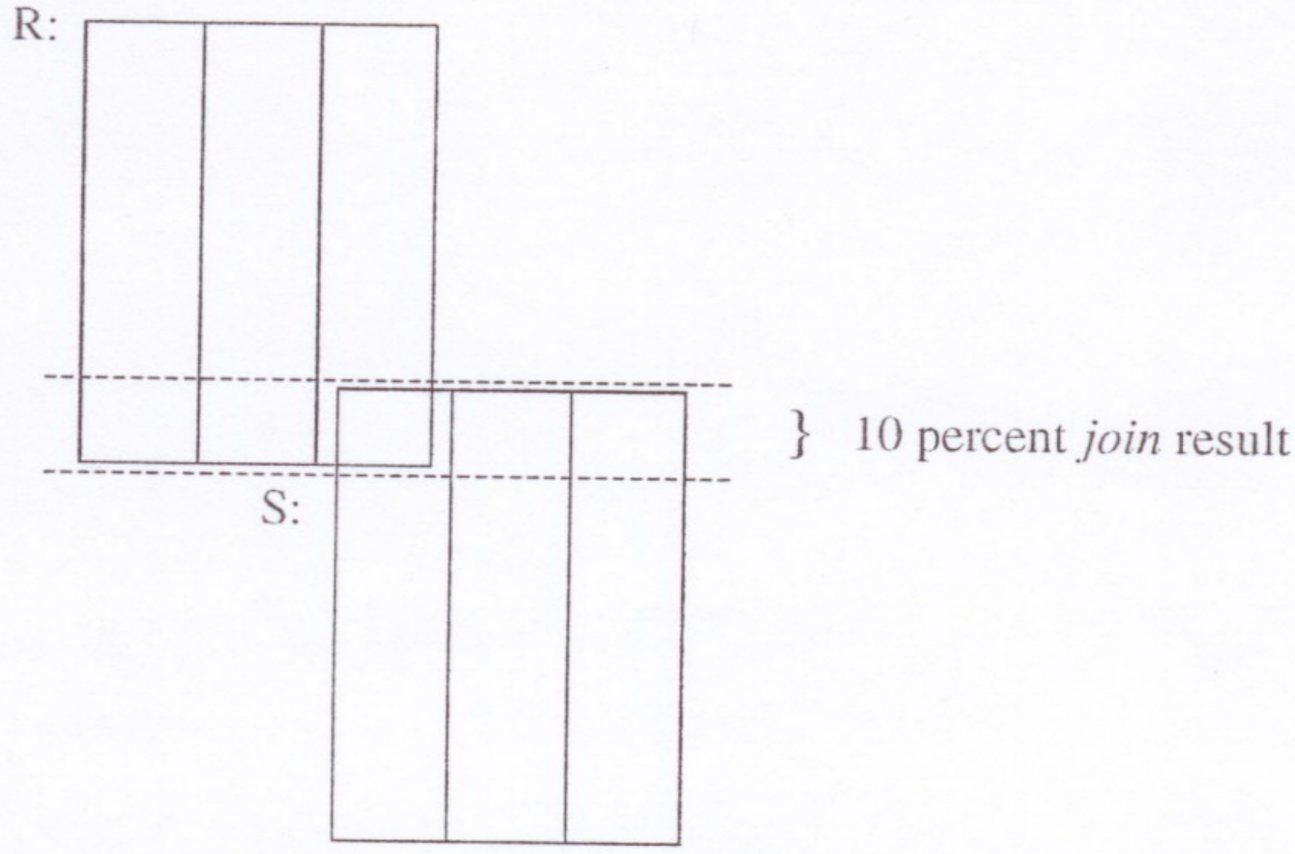
Sample data for input relation R, when n = 10; (simplified arrangement for column 3):

7	3	1
3	8	2
6	4	3
5	1	4
4	10	5
1	6	6
8	9	7
2	5	8
9	7	9
10	2	10

Sample data for relation S, when n = 10:

10	5	9
11	9	8
12	1	2
13	3	7
14	8	1
15	10	3
16	4	10
17	2	5
18	7	6
19	6	4

The join activity is represented diagrammatically by overlaying R, S thus:



With n = 10, Test (a) thus yields an output relation having a single 5-field tuple as follows:

10 2 10 5 9

With n = 10, Test (b) yields the same 5-field tuple as its result. With n = 100, the output from both tests would consist of ten 5-field tuples.

FIGURE 1: Simplified sample data sets for n = 10

It should be emphasised that the ordering of field 3 of relation R in Figure 1 is for illustrative purposes only. Algorithm to generate the actual randomised data to be used in the benchmark tests is given in the Appendix.

Note that join test (a) will yield faster run-times than join test (b) for software systems that only index on first argument.

3. Data-generation and source programs

A C program to generate a file of test data - (i.e. the input relations R, S of the previous section) - is given in the Appendix, pages A2 - A3. Different DBMS or languages may require different separators between fields. At Essex we have used the following conventions: full stop space for PROLOG; comma space for INGRES; space for C; etc. Files of data can be made available from Essex upon request.

A general-purpose C program to perform *joins* is given on Appendix pages A4 - A6. Clearly, other C strategies are possible. The results of running this and other *join* tests are presented in Section 4.

On page A7 we give the code for the *join* benchmark when programmed in the MEGALOG system, produced by ECRC (Ref. 1). MEGALOG, formerly known as KbProlog, provides a logic programming front-end to a relational database management system. MEGALOG uses the balanced and nested grid (BANG) file strategy for data access.

On pages A8 and A9 we give the code for 'fast' and 'slow' versions of Quintus Prolog. The appellations 'fast' and 'slow' correspond to join tests (a) and (b) respectively and reflect the fact that Quintus Prolog's strategy, of indexing on first argument only, produces much faster run-times for test (a). It will be seen that the style of Prolog programming is based on assertions, rather than on the more usual logic programming representation involving lists. We have deliberately chosen the assertions strategy so as to show Prolog's performance in a favourable light. It will be appreciated that an AI programmer may feel that our representation is out of character, and would prefer a more 'natural' (though slower) version of the code.

On pages A10 - A12 we give join tests (a) and (b) for Kyoto Common LISP, based on arrays. Once again, our style of programming may not appeal to LISP list aficionados, but we choose arrays for the sake of speed.

On pages A13 - A15 we present the loading, running and timing code for an INGRES join program.

4. Benchmark results

The *join* benchmark was originally developed at Essex to help evaluate the performance of a knowledge-base server called the IFS/2. The IFS/2 is an add-on hardware unit which uses novel SIMD-parallel techniques both to store and process information held in data-structures such as sets, relations or graphs. (Refs 2, 3). The IFS/2 is connected to a host computer via a

standard SCSI channel. At Essex, the prototype IFS/2 is connected to a Sun Sparc Workstation.

Because Sun Workstations are so widely available, we have run all our software *join* benchmarks on Sun Sparc platforms. Furthermore, we have specified a 16 Mbyte Sun Sparc running at 24 MHz because this technology datum is in common use. Finally, it is close to the 27 Mbytes of semiconductor CAM and 25 MHz clock rate of the IFS/2.

The following software systems have so far been evaluated using the *join* benchmark:

1. A general-purpose C program.
2. MEGALOG (formerly known as KbProlog).
3. Quintus PROLOG.
4. Kyoto Common LISP.
5. The INGRES relational DBMS.
6. Another well-known relational DBMS, labelled X in Figure 2 for reasons of commercial confidentiality.

The source code for each of these programs is discussed in Section 3 and the Appendix. The C, MEGALOG, INGRES and X systems each gave approximately equal run-times for join tests (a) and (b), indicating that the indexing strategies for these four programs were not sensitive to the position of the join fields. However, tests (a) and (b) gave run-times which differed by three orders of magnitude in the case of LISP and two orders of magnitude in the case of PROLOG. For example, Quintus PROLOG yielded the following run-times, measured in seconds:

Input relation cardinality, n	join test (a)	join test (b)
1,000	0.1 secs	18.867 secs
3,375	0.317	219.284
8,000	0.767	1,231.317
15,625	1,483	too long for comfort
27,000	2.533	too long for comfort
42,875	3.867	too long for comfort
64,000	6.100	too long for comfort

In Figure 2 we have plotted the average of the times (a) and (b), in the case of Quintus PROLOG and Kyoto Common LISP.

Figure 2 shows elapsed times in seconds versus relation cardinality (i.e. number of tuples in each input relation), plotted on a log/log scale. For comparison, we also plot the *join* times for

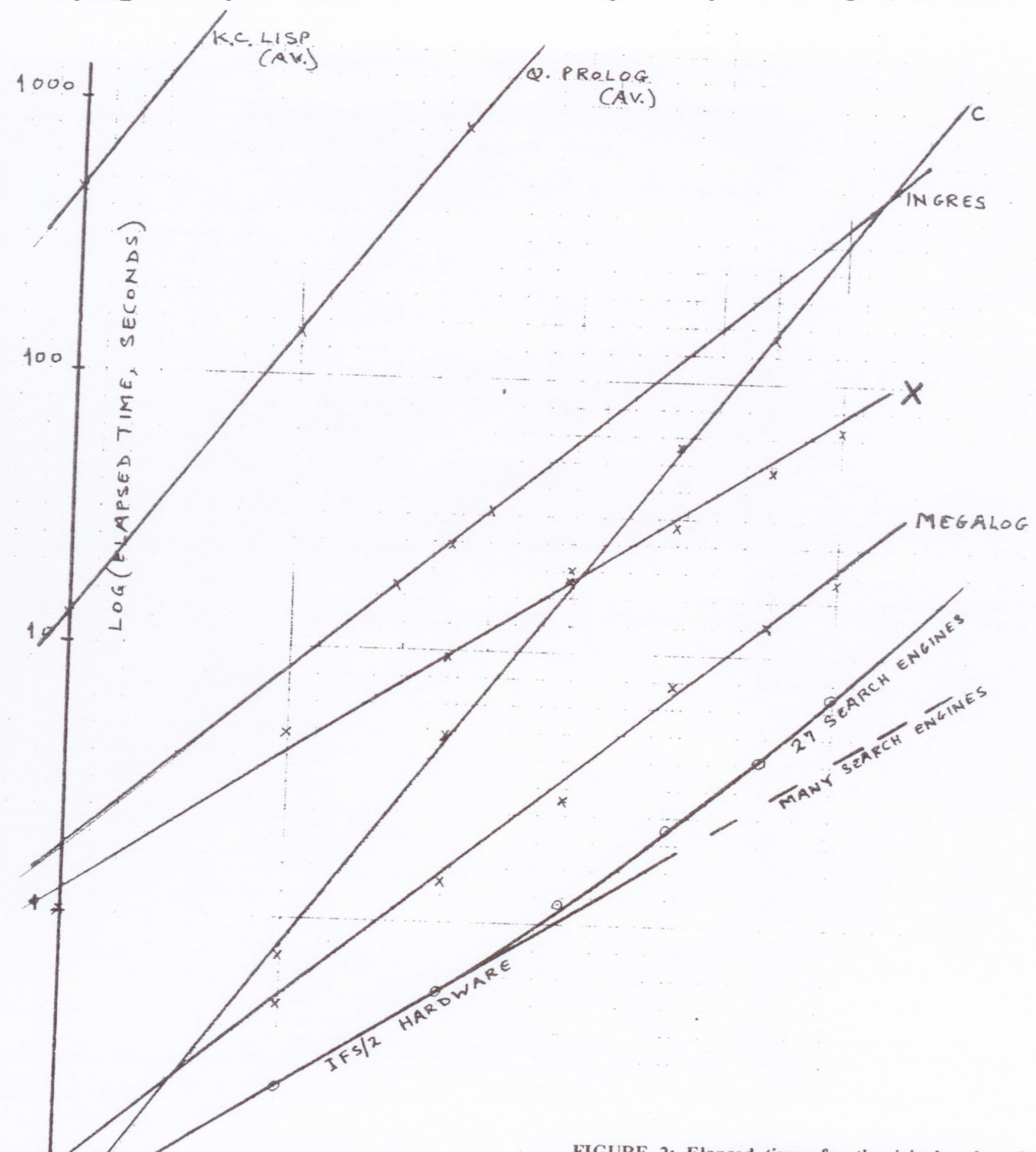


FIGURE 2: Elapsed times for the *join* benchmark, plotted on a log-log scale for values of n ranging from 1000 to 64,000.

the IFS/2 prototype unit. The present IFS/2 is configured to contain 27 SIMD search modules; Figure 2 also shows the performance for an IFS/2 containing an infinite number of search modules.

5. Conclusions

Use of the *join* benchmark for comparing DBMS and KBS systems is an on-going task: more results will be added to Figure 2 in due course. The purpose of this present Report is to introduce the benchmark and encourage its use by others.

As far as conventional software is concerned, Figure 2 supports the general view that the common AI implementation languages are unlikely to be efficient at manipulating structures containing realistic volumes of data. In contrast, the MEGALOG system appears to perform very well.

As far as novel hardware is concerned, it is seen from Figure 2 that the IFS/2 may be expected to perform whole-structure operations between 2 and 5,000 times faster than conventional software. Note that the times given in Figure 2 for the IFS/2 hardware are known to be capable of improvement; the prototype design is currently being refined.

The following general observations may also be made about the IFS/2:

(a) For small relations, the IFS/2 hardware may not be efficient. The break-even point between IFS/2 hardware and conventional software depends on the nature of the whole-structure operation and the particular software system being evaluated. For example, Figure 2 shows that the hand-coded C program performs joins faster than the IFS/2 hardware for relations smaller than about 1000 tuples. The cross-over points for the other software systems of Figure 2 lie below 100 tuples. These conclusions are in line with our measurements of the IFS/2 when supporting the CLIPS Production Rule System (Ref.4). In Ref. 4 it is shown that the IFS/2 may speed up production systems by three orders of magnitude if the number of initial facts is greater than about 10,000. Conversely, use of the IFS/2 actually slows down performance for production systems having fewer than about 100 initial facts.

(b) For the particular *join* tests of Figure 2, there is no advantage in having more than 27 search modules in the IFS/2 hardware for cardinalities under about 3000 tuples. Above that size, the IFS/2 performance curve can be kept linear by adding more search modules.

6. Acknowledgements

It is a pleasure to acknowledge the contribution of other members of the IFS team at Essex. Particularly, Neil Dewhurst and Chetan Mistry helped to obtain some of the results of Figure 2. The work described in this paper has been supported by SERC grants GR/F/06319, GR/F/61028 and GR/G/30867.

7. References

1. **J. Bocca, M. Dahmen, G. Macartney, P. Pearson and A. Berthier,** "KB-Prolog User Guide", *January 1990. Available from the European Computer Industry Research Centre (ECRC), Arabellastrasse 17, D-8000, Munchen 81, Germany.*

2. **S. H. Lavington, J. M. Emby, A. J. March, E. E. James and M. J. Lear,** "A modularly Extensible Scheme for Exploiting Data Parallelism". *Presented at the Third International Conference on Transputer Applications, Glasgow, 1991. Published in Applications of Transputers 3, IOS Press 1991, pages 620-625.*

3. **S. H. Lavington, M. E. Waite, J. Robinson and N. E. J. Dewhurst,** "Exploiting parallelism in primitive operations on bulk data types". *Proceedings on PARLE-92, the Conference on Parallel Architectures and Languages Europe, Paris, June 1992. Published by Springer-Verlag as LNCS 605, pages 893-908.*

4. **S. H. Lavington, C. J. Wang, N. Kasabov and S. Lin,** "Hardware support for data parallelism in production systems". *To be presented at the Third International Workshop in VLSI for Neural Networks and Artificial Intelligence, Oxford, September 1992.*

APPENDIX: Program listings

Contents

```
/* Usage: random seed n1 n2 n3 filename.
   Generates file 'filename1.trip' containing n1 randomly-generated
   triples and 'filename2.trip' containing n2 triples, s.t. joining
   on third column of first and first column of second produces
   n3 responders.
*/

#include <stdio.h>

main (int argc, char *argv[])
{
        int     seed,
                n1,
                n2,
                n3,
                no_trips,
               *a1,
               *a2,
               *a3,
                swap_place_1,
                swap_place_2,
                temp;

register int    i;

char    *f_name;

FILE    *fp;

seed = atoi(argv[1]);
srand (seed);

n1 = atoi(argv[2]);
n2 = atoi(argv[3]);
n3 = atoi(argv[4]);

a1 = (int *)malloc((n1+n2)*sizeof(int));
a2 = (int *)malloc((n1+n2)*sizeof(int));
a3 = (int *)malloc((n1+n2)*sizeof(int));

for (i=0; i<n1; i++)
{
        a1[i] = i+1;
        a2[i] = i+1;
        a3[i] = i+1;
}

for (i=0; i<n2; i++)
{
        a1[i+n1] = i+1+n1-n3;
        a2[i+n1] = i+1;
        a3[i+n1] = i+1;
}

for (i=0; i<n1+n2; i++)
```

```
        {
        /* swap column 1 */

                swap_place_1 = rand()%n1;
                swap_place_2 = rand()%n1;

                temp = a1[swap_place_1];
                a1[swap_place_1] = a1[swap_place_2];
                a1[swap_place_2] = temp;

                swap_place_1 = rand()%n2+n1;
                swap_place_2 = rand()%n2+n1;

                temp = a1[swap_place_1];
                a1[swap_place_1] = a1[swap_place_2];
                a1[swap_place_2] = temp;

                swap_place_1 = rand()%(n1+n2);
                swap_place_2 = rand()%(n1+n2);

                temp = a2[swap_place_1];
                a2[swap_place_1] = a2[swap_place_2];
                a2[swap_place_2] = temp;

                swap_place_1 = rand()%n1;
                swap_place_2 = rand()%n1;

                temp = a3[swap_place_1];
                a3[swap_place_1] = a3[swap_place_2];
                a3[swap_place_2] = temp;

                swap_place_1 = rand()%n2+n1;
                swap_place_2 = rand()%n2+n1;

                temp = a3[swap_place_1];
                a3[swap_place_1] = a3[swap_place_2];
                a3[swap_place_2] = temp;
}

f_name = (char *)malloc(strlen(argv[5])+7);
        strcpy (f_name, argv[5]);
        strcpy (f_name+strlen(argv[5]), "1.trip\0x0");
        fp = fopen (f_name, "w");
        for (i=0; i<n1; i++)
                fprintf (fp, "%d. %d. %d.\n", a1[i], a2[i], a3[i]);
        fclose (fp);

        strcpy (f_name+strlen(argv[5]), "2.trip\0x0");
        fp = fopen (f_name, "w");
        for (i=n1; i<n1+n2; i++)
                fprintf (fp, "%d. %d. %d.\n", a1[i], a2[i], a3[i]);
        fclose (fp);
}
```

```
#include "stdio.h"

#define REL_ENTRY       struct rel_entry

struct rel_entry
{
        int                     e1,
                                e2,
                                e3,
                                e4,
                                e5;
        REL_ENTRY              *next;
} *r1_start = NULL, *r2_start = NULL;


read_rels (char *f_name1, char *f_name2)
{
        FILE                   *fp;
        REL_ENTRY              *r_temp;
        int                     fs_report;

        fp = fopen (f_name1, "r");
        r1_start = (REL_ENTRY *)malloc(sizeof(REL_ENTRY));
        r_temp = r1_start;
        fs_report = fscanf (fp, "%d %d %d", &(r_temp->e1),
                &(r_temp->e2), &(r_temp->e3));
        while (fs_report!=EOF)
        {
                r_temp->next = (REL_ENTRY *)malloc(sizeof(REL_ENTRY));

                fs_report = fscanf (fp, "%d %d %d", &(r_temp->next->e1),
                        &(r_temp->next->e2), &(r_temp->next->e3));
                if (fs_report==EOF)
                {
                        free (r_temp->next);
                        r_temp->next = NULL;
                }
                else
                        r_temp = r_temp->next;
        }
        fclose (fp);

        fp = fopen (f_name2, "r");
        r2_start = (REL_ENTRY *)malloc(sizeof(REL_ENTRY));
        r_temp = r2_start;
        fs_report = fscanf (fp, "%d %d %d", &(r_temp->e1),
                &(r_temp->e2), &(r_temp->e3));
        while (fs_report!=EOF)
        {
                r_temp->next = (REL_ENTRY *)malloc(sizeof(REL_ENTRY));

                fs_report = fscanf (fp, "%d %d %d", &(r_temp->next->e1),
                        &(r_temp->next->e2), &(r_temp->next->e3));
                if (fs_report==EOF)
                {
                        free (r_temp->next);
                        r_temp->next = NULL;
                }
                else
                        r_temp = r_temp->next;
        }
        fclose (fp);
}
```

```
#define I_ENTRY          struct i_entry

struct i_entry
{
        int              entry,
                         position;
        I_ENTRY         *next;
} *i_start = NULL;

I_ENTRY         *read_index ()
{
        FILE            *index;
        I_ENTRY         *i_temp;
        int              fs_report;

        index = fopen ("index", "r");
        i_start = (I_ENTRY *)malloc(sizeof(I_ENTRY));
        i_temp = i_start;

        fs_report = fscanf (index, "%d %d", &(i_temp->entry),
                &(i_temp->position));
        while (fs_report!=EOF)
        {
                i_temp->next = (I_ENTRY *)malloc(sizeof(I_ENTRY));

                fs_report = fscanf (index, "%d %d", &(i_temp->next->entry),
                        &(i_temp->next->position));
                if (fs_report==EOF)
                {
                        free (i_temp->next);
                        i_temp->next = NULL;
                }
                else
                        i_temp = i_temp->next;
        }

        fclose (index);
}


main (int argc, char *argv[])
{
        I_ENTRY         *i_temp;
        REL_ENTRY       *r1_next,
                        *r2_next,
                        *r3_start = NULL,
                        *r3_last = NULL,
                        *r3_new;
register int     i;

read_index();
read_rels(argv[1], argv[2]);

r1_next = r1_start;
while (r1_next!=NULL)
{
        i_temp = i_start;
        while (i_temp!=NULL&&i_temp->entry<=r1_next->e3)
        {
                if (i_temp->entry==r1_next->e3)
                {
                        r2_next = r2_start;
                        for (i=0; i<i_temp->position-1; i++)
                                r2_next = r2_next->next;

                        r3 new = (REL ENTRY *)malloc(sizeof(REL ENTRY));
```

```
                r3_new->e1 = r1_next->e1;
                r3_new->e2 = r1_next->e2;
                r3_new->e3 = r1_next->e3;
                r3_new->e4 = r2_next->e2;
                r3_new->e5 = r2_next->e3;

                if (r3_start==NULL)
                        r3_start = r3_new;
                if (r3_last!=NULL)
                        r3_last->next = r3_new;
                r3_last = r3_new;

        }
                        i_temp = i_temp->next;
                }

                r1_next = r1_next->next;
        }
}
```

```
run(F) :- user_cpu_time(S),
          X isr seta :*: setb where seta^field3 == setb^field1,
          user_cpu_time(F).
init_large :-
        seta <=> [integer(field1,4,'-'),integer(field2,4,'-'),integer(field3,4,'+')
        setb <=> [integer(field1,4,'+'),integer(field2,4,'-'),integer(field3,4,'-')

init_small :-
        seta <=> [integer(field1,2,'-'),integer(field2,2,'-'),integer(field3,2,'+')]
        setb <=> [integer(field1,2,'+'),integer(field2,2,'-'),integer(field3,2,'-')]

go1(N) :-
  open('file1',read,File),
  aloop(N,File),
  close(File).

aloop(0,F).
aloop(X,File) :- read(A,File),read(B,File),read(C,File),
                 ins_tup(seta(A,B,C)),
                 N is X - 1,
                 aloop(N,File).

go2(N) :-
  open('file2',read,File),
  bloop(N,File),
  close(File).

bloop(0,F).
bloop(X,File) :- read(A,File),read(B,File),read(C,File),
                 ins_tup(setb(A,B,C)),
                 N is X - 1,
                 bloop(N,File).
```

:58 1992 quintus_fast Page 1 ~(join Test(a)).

```
run(S,F) :- statistics(runtime,S),join(_),statistics(runtime,F).
join(_) :-
                    seta(A,B,C),
                    setb(C,D,E),
                    assert(int(A,B,C,D,E)),
                    fail.
join(_).


go1(N) :-
  see('file1'),
  aloop(N),
  seen.


go2(N) :-
  see('file2'),
  bloop(N),
  seen.

aloop(0).
bloop(0).
aloop(X) :- read(A),read(B),read(C),assertz(seta(A,B,C)), N is X - 1, aloop(N).
bloop(X) :- read(A),read(B),read(C),assertz(setb(A,B,C)), N is X - 1, bloop(N).

g(S,F,N,R) :- go1(N),go2(N),run(S,F),abolish(int/5).
x(S,F,N,R) :- go1(N),go2(N),run(S,F).
```

:58 1992 quintus_slow Page 1 ~(join test(4)).

```
run(S,F) :- statistics(runtime,S),join(_),statistics(runtime,F).
join(_) :-
                    setb(C,D,E),
                    seta(A,B,C),
                    assert(int(A,B,C,D,E)),
                    fail.
join(_).


go1(N) :-
  see('file1'),
  aloop(N),
  seen.

go2(N) :-
  see('file2'),
  bloop(N),
  seen.

aloop(0).
bloop(0).
aloop(X) :- read(A),read(B),read(C),assertz(seta(A,B,C)), N is X - 1, aloop(N).
bloop(X) :- read(A),read(B),read(C),assertz(setb(A,B,C)), N is X - 1, bloop(N).

g(S,F,N,R) :- go1(N),go2(N),run(S,F),abolish(int/5).
x(S,F,N,R) :- go1(N),go2(N),run(S,F).
```

A 10.

:58 1992 lisp_fast Page 1 ~ (join test (a)).

```
(defun run (S)
        (get-data-a "file1" S)
        (get-data-b "file2" S)
        (setq set-j (make-array '(1216001 5)))
        (print (get-internal-run-time))
        (join S)
        (print (get-internal-run-time))
)

(defun join (S)
        (do ((j 1 (+ j 1))
             (n S ))
             ((> j n))
             (let ((X (aref set-a j 2)))
                 (if (null (aref set-b X 0))
                     ()
                     (setf (aref set-j X 0) (aref set-a j 0)
                           (aref set-j X 1) (aref set-a j 1)
                           (aref set-j X 2) (aref set-a j 2)
                           (aref set-j X 3) (aref set-b X 1)
                           (aref set-j X 4) (aref set-b X 2))
                 )
             )
        )
)

(defun get-data-a (file f-l)
        (let ((input-stream (open file :direction :input)))
             (setq set-a (make-array '(64001 3)))
             (do ((j 0 (+ j 1))
                  (n f-l ))
                  ((= j n))
                  (let ((X (read input-stream nil '|the end|))
                        (Y (read input-stream nil '|the end|))
                        (Z (read input-stream nil '|the end|)))
                       (setf (aref set-a X 0) X)
                       (setf (aref set-a X 1) Y)
                       (setf (aref set-a X 2) Z)))
             (close input-stream)))

(defun get-data-b (file f-l)
        (let ((input-stream (open file :direction :input)))
             (setq set-b (make-array '(121601 3)))
             (do ((j 0 (+ j 1))
                  (n f-l ))
                  ((= j n))
                  (let ((X (read input-stream nil '|the end|))
                        (Y (read input-stream nil '|the end|))
                        (Z (read input-stream nil '|the end|)))
                       (setf (aref set-b X 0) X)
                       (setf (aref set-b X 1) Y)
                       (setf (aref set-b X 2) Z)))
             (close input-stream)))
```

A 11.

:58 1992 lisp_slow Page 1 ~ (join test (4)).

```
(defun run (S)
        (get-data-a "sfile1" S)
        (get-data-b "sfile2" S)
        (setq set-j (make-array '(1901 5)))
        (print (get-internal-run-time))
        (join S)
        (print (get-internal-run-time))
)

(defun join (S)
        (do ((j 1 (+ j 1))
             (n S ))
             ((> j n))
             (let ((X (aref set-a j 2)))
                 (do ((i 900 (+ i 1))
                  (nn (+ S 900)))
                 ((> i nn))
                 (let ((Y (aref set-b i 0)))
                     (if (null Y)
                         ()
                         (if (eql X Y)
                             (setf (aref set-j X 0) (aref set-a j 0)
                                   (aref set-j X 1) (aref set-a j 1)
                                   (aref set-j X 2) (aref set-a j 2)
                                   (aref set-j X 3) (aref set-b i 1)
                                   (aref set-j X 4) (aref set-b i 2))
                         )
                     )
                 )
             )
        )
)

(defun get-data-a (file f-l)
        (let ((input-stream (open file :direction :input)))
             (setq set-a (make-array '(1001 3)))
             (do ((j 0 (+ j 1))
                  (n f-l ))
                  ((= j n))
                  (let ((X (read input-stream nil '|the end|))
                        (Y (read input-stream nil '|the end|))
                        (Z (read input-stream nil '|the end|)))
                       (setf (aref set-a X 0) X)
                       (setf (aref set-a X 1) Y)
                       (setf (aref set-a X 2) Z)))
             (close input-stream)))

(defun get-data-b (file f-l)
        (let ((input-stream (open file :direction :input)))
             (setq set-b (make-array '(1901 3)))
             (do ((j 0 (+ j 1))
                  (n f-l ))
                  ((= j n))
                  (let ((X (read input-stream nil '|the end|))
                        (Y (read input-stream nil '|the end|))
```

```
                    (Z (read input-stream nil '|the end|)))
                  (setf (aref set-b X 0) X)
                  (setf (aref set-b X 1) Y)
                  (setf (aref set-b X 2) Z)))
          (close input-stream)))
```

```
/*-----------------
  | Timing INGRES |
  -----------------*/

/*
  This file contains the information used in timing INGRES.
  It contains the information for loading in the data, the join
  query used on the data, and how INGRES was timed for the query.
*/

/*
  Table ONE and TWO both consist of three columns of integer values,
  randomly generated. The data for the tables is specified in
  "/ufs/ingres/100.1.sig" and "/ufs/ingres/100.2.sig" and are of the
  following format :

  23, 6, 84
  64, 13, 71
  9, 46, 29
  12, 22, 55

  etc, etc.

  All columns are of type smallint.

*/

create table ONE (
        cola    smallint,
        colb    smallint,
        colc    smallint
        );

copy table ONE (
        cola    = c0,
        colb    = c0,
        colc    = c0
        )
        from '/ufs/msc2/mistc/ingres/100.1.sig';

create table TWO (
        cola    smallint,
        colb    smallint,
        colc    smallint
        );

copy table TWO (
        cola    = c0,
        colb    = c0,
        colc    = c0
        )
        from '/ufs/msc2/mistc/ingres/100.2.sig';


/*
  The join query is specified in a file called "join" and
  specifies a join on colc of ONE equal to cola of TWO.
  The "\g" is to execute the query.
*/

select *
from ONE,TWO
where ONE.colc = TWO.cola; \g


/*
```

```
  Timing INGRES was done using a 'C' program. Firstly, the database is
  created and the data is loaded. Then timings are done to see how long it
  takes to access ingres (from UNIX command line) without specifying any
  operation, this gives T1 . Then timings are taken with the join query
  inputted from the file "join", which gives T2. To find the time taken,
  (T2 - T1) is calculated.

  The 'C' programming language provides two utility functions, setitimer()
  and getitimer() which allows you to set up the clock and get the time at
  any given point.
*/

#include <stdio.h>
#include <sys/time.h>

main(argc,argv)
int argc;
char *argv[];

{
  struct itimerval      before,      /* structures for holding the time */
                        after,       /* before and after the execution  */
                        *init,
                        *ovalue;


/*
  Allocate memory for timer.
*/

init = (struct itimerval *)malloc(sizeof(struct itimerval));

/*
  Give clock initial value of 999.999999 seconds
  The clock counts down
*/

init->it_interval.tv_sec = 999;
init->it_interval.tv_usec = 999999;
init->it_value.tv_sec = 999;
init->it_value.tv_usec = 999999;

/*
  start the clock.
*/

if (setitimer(ITIMER_REAL,init,ovalue) == -1) printf("Timer error");

/*
  get the time just before execution
*/

if (getitimer(ITIMER_REAL,&before) == -1) printf("Timer error");

/*
  Execute the the query as specified in "join"
  "sql" starts INGRES, and "temp" is the name of the database
  Dump all result in a file called "out"

  N.B. To do initial timing (T1) remove "< join"
*/

system("sql temp < join > out");

/*
  Get time just after the execution has finished
```

```
   */

   if (getitimer(ITIMER_REAL,&after) == -1) printf("Timer error");

   /*
     print the time of the clock before and after execution
   */

   printf("\nTime before = %d:%d      Time after %d:%d\n",
            before.it_value.tv_sec, before.it_value.tv_usec,
            after.it_value.tv_sec, after.it_value.tv_usec);
}

/*
  Chetan Mistry 28/4/92
*/
```