The Wisconsin Benchmark: Past, Present, and Future

David J. DeWitt
Computer Sciences Department
University of Wisconsin

1.0. Introduction

In 1981 as we were completing the implementation of the DIRECT database machine [DEWI79, BORA82], attention turned to evaluating its performance. At that time no standard database benchmark existed. There were only a few application-specific benchmarks. While application-specific benchmarks measure which database system is best for a particular application, it was very difficult to understand them. We were interested in a benchmark to measure DIRECT's speedup characteristics. Thus, we set out to develop a benchmark that could be used to evaluate DIRECT both relative to itself and relative to the "university" version of Ingres.

The result of this effort was a benchmark for relational database systems and machines that has become known as the *Wisconsin Benchmark* [BITT83]. The benchmark was designed with two objectives in mind. First, the queries in the benchmark should test the performance of the major components of a relational database system. Second, the semantics and statistics of the underlying relations should be well understood so that it is easy to add new queries and to their behavior.

We never expected this benchmark to become as popular as it did. In retrospect, the reasons for this popularity were only partially due to its technical quality. The primary reason for its success was that it was the first evaluation containing impartial measures of real products. By actually identifying the products by name, the benchmark triggered a series of "benchmark wars" between commercial database products. With each new release, each vendor would produce a new set of numbers claiming superiority. With some vendors releasing their numbers, other vendors were obliged to produce numbers for their own systems. So the benchmark quickly became a standard which customers knew about and wanted results for. In retrospect, had the products not been identified by name, there would have been no reason for the vendors to react the way they did, and the benchmark would most likely have simply been dismissed as an

academic curiosity. We did not escape these wars completely unscarred. In particular, the CEO of one of the companies repeatedly contacted the chairman of the Wisconsin Computer Sciences Department complaining that we had not represented his product fairly.

The benchmark changed the database marketplace in several ways. First, by pointing out the performance warts of each system, vendors were forced to significantly improve their systems in order to remain competitive. Consequently, the significant performance differences among the various products observed in 1983 gradually disappeared. While the vendors undoubtedly wasted some time fighting these wars, overall the customers clearly benefited from them.

The second major effect of developing the Wisconsin benchmark was to spur the development of the Datamation benchmark [ANON88] by a large group lead by Jim Gray. From their viewpoint (in 1984), the Wisconsin benchmark had little to do with "real" database applications - something the Datamation benchmark set out to rectify. Seven years later, the DebitCredit transaction of the Datamation benchmark has replaced the Wisconsin benchmark as the standard for relation products despite the fact that it fails to test a number of the major components of a relational database system. For example, the Datamation benchmark does not detect whether a system even has a query optimizer, let alone, determine whether nested-loops is the only join method provided. Furthermore, most vendors have ignored the scan and sort components of the Datamation benchmark, implementing only the DebitCredit portion, diluting the value of the original benchmark significantly. Why then is the benchmark so popular? Ignoring its technical merits, one reason is that it reduces each system to one magic number - its "tps" rating (which like a "sound bite" is attractive to the marketing types of the world) even though the benchmark says nothing about the performance of a system on applications that are not debit-credit.

Like the early relational products, the Wisconsin benchmark is not, however, without its flaws [BITT88]. While it is missing a number of easily added tests such as bulk updates, its most critical failing is that, being a single user benchmark, it does not test critical features required for real-world applications. The most important features missing from the Wisconsin benchmark are tests of the concurrency control and recovery subsystems of a database system. Thus, the benchmark cannot distinguish a system that supports only relational-level locking from one that provides tuple-level locking. While the benchmark has been widely criticized for its single-user nature, this criticism is not really justified. Immediately after completing the single-user benchmark we undertook the development of a multiuser version of the benchmark. As the result of technical disagreements, this produced two competing multiuser benchmarks [BORA84,

BITT85] neither of which ever attracted widespread interest or use. In a number of ways both these benchmarks represent a much more scientific approach to designing a multiuser benchmark than did the debit-credit benchmark. Why is it that neither of these benchmarks gained the same popularity as the Wisconsin benchmark? There seem to be two possible explanations. The first is that, like the Wisconsin benchmark, neither reduced each system to a single number, making it more difficult to directly compare two systems (regardless of the superficiality of the comparison). Second, after our experiences with the Wisconsin benchmark, both benchmarks carefully avoided initiating a new benchmark war among the vendors. They considered the multiuser performance of only a single system. This enabled the vendors to simply ignore the results because nobody was provided with ammunition to keep the war going. In retrospect, being gun shy was a mistake. It is now clear that comparisons of actual systems are important if you want a benchmark to become popular.

The remainder of this chapter is organized as follows. Section 2 describes the Wisconsin benchmark, summarizing the key results obtained using the benchmark from [BITT83]. It also describes the weaknesses of the benchmark. While the Wisconsin benchmark is no longer widely used to evaluate single processor relational database systems, it is again finding applicability in evaluating the performance of ad-hoc queries on parallel database systems [DEWI87, DEWI88, ENGL89a, DEWI90]. It has come full circle to its original goal of evaluating highly parallel database machines. As demonstrated in Section 3, the benchmark is well suited for such applications. It is straightforward to use the benchmark to measure the speedup, scaleup, and sizeup characteristics of a parallel database system. This application of the benchmark is illustrated with results obtained for the Gamma database machine. Our conclusions are contained in Section 4.

2.0. An Overview of the Wisconsin Benchmark

This section begins with a description of the database that forms the basis of the Wisconsin benchmark, indicating how the relations that form the benchmark can be scaled to a wide range of sizes. Next, the benchmark queries are described, summarizing the results presented originally in [BITT83]. While the original numbers are no longer of any value, it is interesting to reflect on some of the technical limitations that the benchmark uncovered in the early relational products. Finally, we comment on what we did right and what we did wrong in the designing the benchmark.

2.1. The Wisconsin Benchmark Relations

The development of the Wisconsin benchmark began with a database design flexible enough to allow straightforward specification of a wide range of retrieval and update queries. One of the early decisions was to use synthetically generated relations instead of empirical data from a real database. A problem with empirical databases is that they are difficult or impossible to scale. A second problem is that the values they contain are not flexible enough to permit the systematic benchmarking of a database system. For example, with empirical data it is very difficult to specify a selection query with a 10% or 50% selectivity factor or one that retrieves precisely one thousand tuples. For queries involving joins, it is even harder to model selectivity factors and build queries that produce results or intermediate relations of a certain size. An additional shortcoming of empirical data (versus "synthetic" data) is that one has to deal with very large amounts of data before it can be safely assumed that the data values are randomly distributed. By building a synthetic database, random number generators can be used to obtain uniformly distributed attribute values, and yet keep the relation sizes tractable.

The database is designed so that one can quickly understand the structure of the relations and the distribution of each attribute value. Consequently, the results of the benchmark queries are easy to understand and additional queries are simple to design. The attributes of each relation are designed to simplify the task of controlling selectivity factors in selections and joins, varying the number of duplicate tuples created by a projection, and controlling the number of partitions in aggregate function queries. It is also straightforward to build an index (primary or secondary) on some of the attributes, and to reorganize a relation so that it is clustered with respect to an index.

2.1.1. Overview of the Original Benchmark Relations

The original benchmark was composed of three relations, one with 1,000 tuples (named ONEKTUP) and two others each with 10,000 tuples (named TENKTUP1 and TENKTUP2). Each relation was composed of the thirteen integer attributes and three 52 byte string attributes. While two byte integers were used in the original benchmark (because DIRECT was implemented using PDP 11 processors which did not support four byte integers), fairly early on most users of the benchmark switched to using four byte integers. Thus, assuming no storage overhead, the length of each tuple is 208 bytes. The SQL statement to create the TENKTUP1 relation is shown in Figure 1. The same basic schema is used to create other relations (e.g. a second 10,000 tuple relation or one containing a million tuples) .

```
CREATE TABLE TENKTUP1
(  unique1        integer NOT NULL,
   unique2        integer NOT NULL PRIMARY KEY,
   two            integer NOT NULL,
   four           integer NOT NULL,
   ten            integer NOT NULL,
   twenty         integer NOT NULL,
   hundred        integer NOT NULL,
   thousand       integer NOT NULL,
   twothous       integer NOT NULL,
   fivethous      integer NOT NULL,
   tenthous       integer NOT NULL,
   odd100         integer NOT NULL,
   even100        integer NOT NULL,
   stringu1       char(52) NOT NULL,
   stringu2       char(52) NOT NULL,
   string4        char(52) NOT NULL
)
```

Figure 1: SQL Create Statement for the Original 10,000 Tuple Relation

As discussed above, in designing the structure of the base relations for the benchmark, one goal was to make it easy to understand the semantics of each attribute so that a user could extend the benchmark by adding new queries. Table 1 summarizes the semantics of each attribute for the 10,000 tuple relation.

Attribute Name	Range of Values	Order	Comment
unique1	0-9999	random	candidate key
unique2	0-9999	random	declared key
two	0-1	cyclic	0,1,0,1,...
four	0-3	cyclic	0,1,2,3,0,1,...
ten	0-9	cyclic	0,1,...,9,0,1,...
twenty	0-19	cyclic	0,1,...,19,0,1,...
hundred	0-99	cyclic	0,1,...,99,0,1,...
thousand	0-999	cyclic	0,1,...,999,0,1,...
twothous	0-1999	cyclic	0,1,...,1999,0,1,...
fivethous	0-4999	cyclic	0,1,...,4999,0,1,...
tenthous	0-9999	cyclic	0,1,...,9999,0,1,...
odd100	1-99	cyclic	1,3,5,...,99,1,3,...
even100	2-100	cyclic	2,4,...,100,2,4,...
stringu1	-	random	candidate key
stringu2	-	cyclic	candidate key
string4	-	cyclic	

Table 1: Attribute Specifications of Original Wisconsin Benchmark Relations

The values of the unique1 and unique2 attributes are uniformly distributed unique random numbers in the range 0 to MAXTUPLES-1, where MAXTUPLES is the cardinality of the relation. Thus, both "unique1" and "unique2" are candidate keys but "unique2" is used whenever a declared key must be specified. When tests are conducted using indices, unique2 is used to build a

clustered unique index, unique1 is used to build a non-clustered unique index, and hundred is used to build a non-clustered, non-unique index.

After the "unique1" and "unique2" attributes come a set of integer-valued attributes that assume non-unique values. The main purpose of these attributes is to provide a systematic way to model a wide range of selectivity factors. Each attribute is named after the range of values the attribute assumes. For example, the "four" attribute assumes a range of values from 0 to 3. Thus, the selection predicate "x.four=2" will have a 25% selectivity, regardless of the size of the relation to which it is applied. As another example, the "hundred" attribute can be used to create 100 partitions in an aggregate query if a "group by hundred" clause is used.

Finally, each relations has 3 string attributes. Each string is 52 letters long, with three letters (the first, the middle and the last) being varied, and two separating substrings that contain only the letter x. The three significant letters are chosen in the range (A,B,...,V), to allow up to 10,648 (22 * 22 * 22) unique string values. Thus all string attributes follow the pattern:

```
$xxxxxxxxxxxxxxxxxxxxxxxxx$xxxxxxxxxxxxxxxxxxxxxxxx$
              {25 x's}             {24 x's}
```

where "$" stands for one of the letters (A,B,...,V). Clearly, this basic pattern can be modified to provide a wider range of string values (by replacing some of the x's by significant letters). On the other hand, by varying the position of the significant letters, the database designer can also control the cpu time required for string comparisons.

The first two string attributes are string versions of the "unique1" and "unique2" attributes. That is, "stringu1" and "stringu2" may be used as key attributes, and a primary index may be built on "stringu2". For example, in the thousand tuple relation, "ONEKTUP", the stringu2 attribute values are:

```
"Axxxx ... xxxAxxx ... xxxA"
"Bxxxx ... xxxAxxx ... xxxA"
"Cxxxx ... xxxAxxx ... xxxA"
          •••
"Vxxxx ... xxxAxxx ... xxxA"
"Axxxx ... xxxBxxx ... xxxA"
          •••
"Vxxxx ... xxxBxxx ... xxxA"
"Axxxx ... xxxCxxx ... xxxA"
          •••
"Vxxxx ... xxxVxxx ... xxxA"
"Axxxx ... xxxAxxx ... xxxB"
          •••
"Ixxxx ... xxxBxxx ... xxxC"
"Jxxxx ... xxxBxxx ... xxxC"
```

The following two queries illustrate how these string attributes can be utilized. Each query has a selectivity factor of 1%.

```
select * from TENKTUP1
where stringu2 < 'Axxx ... xxxExxx...xxxQ';

select * from TENKTUP1
where  stringu2  between 'Bxxxx ... xxxGxxx ... xxxE'
                     and 'Bxxxx ... xxxLxxx ... xxxA'
```

The "stringu2" variables are initially loaded in the database in the same order in which they were generated, shown above, which is not in sort order. The attribute "stringu1" assumes exactly the same string values as "stringu2" except that their position in the relation is randomly determined. A third string attribute, "string4", assumes only four unique values:

```
"Axxxx ... xxxAxxx ... xxxA"
"Hxxxx ... xxxHxxx ... xxxH"
"Oxxxx ... xxxOxxx ... xxxO"
"Vxxxx ... xxxVxxx ... xxxV"
```

2.1.2. Scaling the Benchmark Relations

Two criticisms of the benchmark relations were that the relations were difficult to scale (make larger) and that the structure of the string attributes were not realistic [BITT88]. While a two megabyte relation was a reasonable choice in 1983 when most mid-range processors had main memories in the megabyte range, the 10,000 tuple relations have become much too small for current single-processor machines, let alone multiprocessor database machines. (However, one still finds people using them - even on machines with 16 or 32 megabytes of memory!)

While scaling most of the original attributes was straightforward, the meaning of certain attributes changed as the relations were scaled. To solve this problem, the specification of the benchmark relations was slightly modified as shown in Table 2.

Attribute Name	Range of Values	Order	Comment
unique1	0-(MAXTUPLES-1)	random	unique, random order
unique2	0-(MAXTUPLES-1)	sequential	unique, sequential
two	0-1	random	(unique1 mod 2)
four	0-3	random	(unique1 mod 4)
ten	0-9	random	(unique1 mod 10)
twenty	0-19	random	(unique1 mod 20
onePercent	0-99	random	(unique1 mod 100)
tenPercent	0-9	random	(unique1 mod 10)
twentyPercent	0-4	random	(unique1 mod 5)
fiftyPercent	0-1	random	(unique1 mod 2)
unique3	0-(MAXTUPLES-1)	random	unique1
evenOnePercent	0,2,4,...,198	random	(onePercent * 2)
oddOnePercent	1,3,5,...,199	random	(onePercent * 2)+1
stringu1	-	random	candidate key
stringu2	-	random	candidate key
string4	-	cyclic	

Table 2: Attribute Specification of "Scalable" Wisconsin Benchmark Relations

The first 6 attributes (unique1 through twentyPercent) remain basically the same except for a couple of minor changes. First, while the values of unique1 continue to be randomly distributed unique values between 0 and MAXTUPLES-1, the values of unique2 are in sequential order from 0 to MAXTUPLES-1. Second, instead of the two, four, ten, and twenty attributes repeating in a cyclic pattern as before, these attributes are now randomly distributed as they are generated by computing the appropriate mod of unique1. More significantly, the hundred through even100 attributes of the original relations have been replaced with a set of attributes that simplify the task of scaling selection queries with a certain selectivity factor. For example, the predicate "twentyPercent = 3" will always return 20% of the tuples in a relation, regardless of the relation's cardinality.

The string attributes have also been completely redesigned to eliminate several of their earlier flaws. Their fixed-length nature was retained so that each disk page contains the same number of tuples. This simplifies the task of understanding the results of a benchmark. The string attributes, stringu1 and stringu2, are the string analogies of unique1 and unique2. Both are candidate keys and can be used in queries just like unique1 and unique2. Both stringu1 and stringu2 consist of seven significant characters from the alphabet ('A'-'Z') followed by 45 x's. The seven significant characters of stringu1 (stringu2) are computed using the following procedure. This procedure converts a unique1 (unique2) value to its corresponding character string representation, padding the resulting string to a length of 7 characters with A's.

```
char *convert(unique)
       int unique;
      { char tmp[7], result[7];
        int i,j, rem, cnt;

        /* first set result string to "AAAAAAA" */
        for  ( i=0 ; i<7 ; i++)   result[i]='A';

        i = 6;  cnt = 0;
         /* convert unique value from right to left into an alphabetic string in tmp
*/
       /*  tmp digits are right justified in tmp                                    */
        while ( (unique > 0) )
              { rem = unique % 26; /* '%' is the mod operator in C */
                tmp[i] = 'A' + rem;
                unique = unique / 26;
                i--; cnt++;
               }
        /* finally move tmp into result, left justifying it */
        for (j=0; j <= cnt; j++, i++) result[j]=tmp[i];
        return  (&result[0])
      }
```

The last string attribute, string4, assumes four values, AAAAxxx..., HHHHxxx..., OOOOxxx..., and VVVVxxx... in a cyclic manner:

```
AAAAxxx...
HHHHxxx...
OOOOxxx...
VVVVxxx...
AAAAxxx...
HHHHxxx...
OOOOxxx...
VVVVxxx...
```

In addition to the information contained in Table 2, in order to generate relations with a range of cardinalities a mechanism is also needed for generating unique1 values. These values must be both unique and in random order. The C procedure in Figure 2, based on an algorithm developed by Susan Englert and Jim Gray [ENGL89b], will efficiently produce such a stream for relations with cardinalities up to 100 million tuples.

```
long      prime, generator;

main (argc, argv)
  int argc; char *argv[];
{
  int     tupCount;          /* number of tuples in result relation       */
  tupCount = atoi (argv[1]);  /* get the desired table size                */
  /* Choose prime and generator values for the desired table size
   */
  if      (tupCount <= 1000)    { generator = 279;  prime = 1009;    }
  else if (tupCount <= 10000)   { generator = 2969; prime = 10007;   }
  else if (tupCount <= 100000)  { generator = 21395;    prime = 100003;     }
  else if (tupCount <= 1000000)                    { generator = 2107;     prime = 10000
  else if (tupCount <= 10000000)                   { generator = 211;      prime = 10000
  else if (tupCount <= 100000000)                  { generator = 21;       prime = 10000
  else { printf("too many rows requested\n"); exit();}

  generate_relation(tupCount);
}

generate_relation (tupCount)
   int tupCount; /* number of tuples in relation */
{  int unique1, i;
   long rand(),  seed;
    seed = generator;
    /* generate values */
    for (i=0;i<tupCount;i++)
      {  seed = rand(seed,(long)tupCount);
         unique1 = (int) seed - 1;
         unique2 = i;
         /* statements to generated other attribute values as per table 1 go here */
         insert into wisconsin_table values  (unique1, unique2, two, four,...);
      }
}

/* generate a unique random number between 1 and limit*/
long rand (seed, limit)
  long seed , limit;
  { do { seed = (generator * seed) % prime; } while (seed > limit);
    return (seed);
  }
```

Figure 2: Skeleton Benchmark Relation Generator

2.2 The Wisconsin Benchmark Query Suite

The suite of benchmark queries was designed to measure the performance of all the basic relational operations including:

1) Selection with different selectivity factors.
2) Projections with different percentages of duplicate attributes.
3) Single and multiple joins.
4) Simple aggregates and aggregate functions.
5) Append, delete, modify.

In addition, for most queries, the benchmark contains two variations: one that can take advantage of a primary, clustered index, and a second that can only use a secondary, non-clustered index.

Typically, these two variations were obtained by using the "unique2" attribute in the first case, and the "unique1" attribute in the second. When no indices have been created, the queries are the same. The benchmark contains a total of 32 queries. The SQL specification of each query is in Appendix I.

Several rules must be followed when running this benchmark in order to insure that the results obtained accurately represent the system being tested. First, the size of the benchmark relations should be at least a factor of 5 larger than the total main memory buffer space available. Thus, if the buffer pool is 4 megabytes, the benchmark relations should each contain 100,000 tuples. If the tests are being performed on a shared-nothing multiprocessor, this sizing task should use the total aggregate buffer space available on all the processors. Second, for each of the 32 queries in the benchmark, the response time for a query must be measured by computing the average elapsed time of several "equivalent" queries. Typically, ten queries are used for the selection, update, and delete tests and four queries are used for the join, aggregate, and projection tests. The queries in each set must alternate between two identical sets of base relations in order to minimize the impact of buffer pool hits on the actual execution times obtained [BITT83].

Elapsed time is used as the performance metric as we have found that more detailed metrics (e.g. CPU time and/or the number of disk I/Os performed) vary unpredictably both among different operating systems and different database systems running on the same operating system. The original Wisconsin benchmark did not incorporate the cost of the software and hardware between tested because the same hardware configuration was used for all the systems (except DIRECT). Scaling the average response time for each query with a price metric would, however, be possible [ANON88]. Clearly such an adjustment is necessary if the systems being compared vary widely in price.

2.2.1. Selection Queries

The speed at which a database system can process a selection operation depends on a number of factors including:

1) The storage organization of the relation.
2) The selectivity factor of the predicate.
3) The hardware speed and the quality of the software.
4) The output mode of the query.

The selection queries in the Wisconsin benchmark explore the effect of each of these four factors. In addition, the impact of three different storage organizations is considered:

1) Sequential (heap) organization.
2) Primary clustered index on the unique2 attribute. (Relation is sorted on unique2 attribute)
3) Secondary, dense, non-clustered indices on the unique1 and onePercent attributes.

These three storage organizations were selected because they are representative of the access methods provided by most relational DBMSs.

The first six selection queries (Queries 1 to 6 in Appendix I) explore the effects of two different selectivity factors (1% and 10%) and three different storage organizations. While our original experiments considered a wider range of selectivity factors, the results indicated that selectivity factors of 1% and 10% produced representative results. These six queries insert their result tuples into a relation. Doing so can affect the response time measured in several ways. First, each time a page of output tuples must be written to disk, the disk heads must be moved from their current position over the source relation to the end of the result relation. While the impact of this head movement is not that significant when the selectivity factor of the predicate is low, it can become significant. For example, with a 50% selectivity factor, for every two pages read, one will be written, causing two random seeks to occur for every 3 pages read at steady state (i.e.,. the buffer pool is full). Second, if the system being tested automatically eliminates duplicate tuples from a result relation, the cost of this duplicate elimination phase can become a significant fraction of the response time of the query.

Query seven selects one tuple using a clustered index, returning the result tuple to the user. The response time for this very simple query provides a good measure of the path length of a system. Query eight quantifies the cost of formatting and displaying tuples on a user's screen. Like query three, query eight has a selectivity factor of 1% and uses a clustered index to retrieve the qualifying tuples. Thus, the difference in the response time of the two queries provides a reasonable estimate of the time to format and display result tuples on a user's screen. Unfortunately, since most modern database systems employ a two process structure, the response times for these two queries also includes the cost of moving data to the user process from the database system process. [EPS87] estimates that the cost to format, transmit, and display each result tuple is about 15 ms.

Some of the most interesting results obtained using these queries in [BITT83] had to do with the relatively poor performance of DIRECT and ORACLE. The design of DIRECT suffered from two serious problems. First, it used message passing extensively to coordinate the processors working in parallel. Second, DIRECT's design attempted to substitute parallelism for indices. As a consequence of these design flaws, DIRECT's performance was significantly worse

than the other systems on both non-indexed and indexed queries. The relatively poor performance of ORACLE made it apparent that the system had some fairly serious problems that needed correction for ORACLE was typically a factor of 5 slower than INGRES and the IDM 500 on most selection queries (it should now be obvious why [BITT83] set off a series of benchmark wars). One other interesting observation had to do with the execution of the 10% non-clustered index selection. Depending on the page size being used, in some cases it is faster to execute this query by scanning the relation sequentially rather than using the index. While some of the query optimizers in 1983 recognized this fact, others did not. Even today (1990) some optimizers fail to optimize this query correctly.

2.2.2. Join Queries

The join queries in the benchmark were designed to study the effect of three different factors:

1) The impact of the complexity of a query on the relative performance of the different database systems.
2) The performance of the join algorithms used by the different systems.
3) The effectiveness of the query optimizers on complex queries.

Three basic join queries are used in the benchmark:

1) **JoinABprime** - a simple join of relations A and Bprime where the cardinality of the Bprime relation is 10% that of the A relation. Thus, if A contains 100,000 tuples, Bprime contains 10,000 tuples. The result relation has the same number of tuples as the Bprime relation.[1]
2) **JoinASelB** - this query is composed of one join and one selection. A and B have the same number of tuples. The selection on B has a 10% selectivity factor, reducing B to the size of the Bprime relation in the JoinABprime query. The result relation for this query has the same number of tuples as the corresponding JoinABprime query. As shown in query 9 in Appendix I, the actual query was formulated with the join operation (`TENKTUP1.unique1 = TENKTUP2.unique1`) preceding the selection operation (`TENKTUP1.unique2 < 1000`) in order to test whether the system has even a rudimentary query optimizer. Surprisingly, some of the early relational products did not and executed the join first.

[1] For each join operation, the result relation contains all the fields of both input relations.

3) **JoinCselASelB** - this query, as shown in Figure 3, is composed of two selections and two joins. Relations A and B contain the same number of tuples. The selections on A and B each have a selectivity factor of 10%. Since each tuple joins with exactly one other tuple, the join of the restricted A and B relations yields an intermediate relation equal in size to its two input relations. This intermediate relation is then joined with relation C, which contains 1/10 the number of tuples there are in A and B. For example, assume A and B contain 100,000 tuples. The relations resulting from selections on A and B will each contain 10,000 tuples. Their join results in an intermediate relation of 10,000 tuples. This relation will be joined with a C relation containing 10,000 tuples and the result of the query will contain 10,000 tuples.

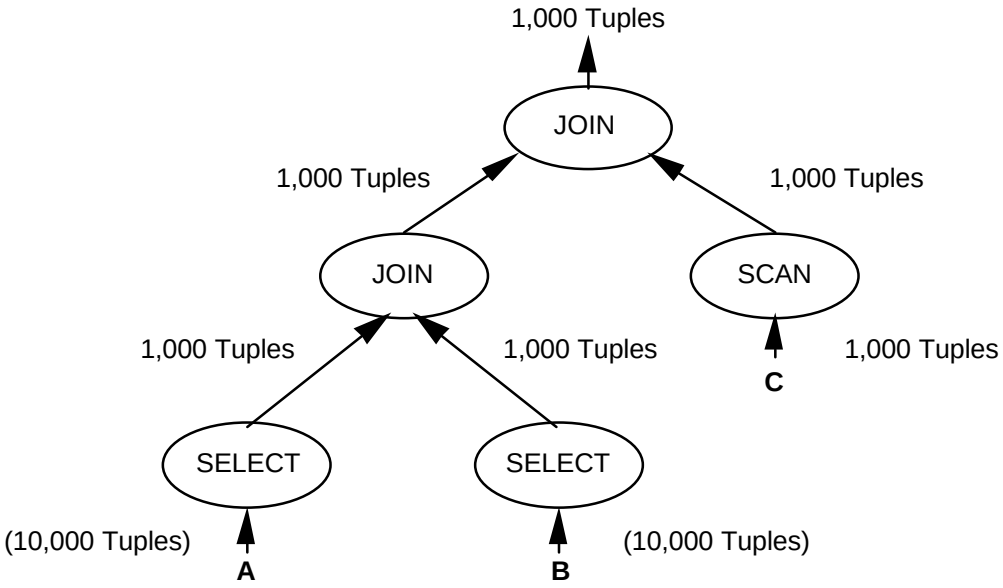

Figure 3: JoinCselAselB

There are three versions of each of these three queries corresponding to either no index, a clustered index, or a non-clustered index on the join attribute. The SQL specifications for these queries are contained in queries 9 through 17 of Appendix I. Queries 11, 14, and 17, warrant, however, additional explanation. In the original version of the benchmark, each of these queries specified selections on both the TENKTUP1 and TENKTUP2 relations. For example, the original "where" clause for query 11 was:

```
WHERE (ONEKTUP.unique2 = TENKTUP1.unique2)
  AND (TENKTUP1.unique2 = TENKTUP2.unique2)
  AND (TENKTUP1.unique2 < 1000)
  AND (TENKTUP2.unique2 < 1000)
```

To further test the capabilities of the query optimizer, in the revised benchmark the selection "TENKTUP2.unique2 < 1000" has been dropped for each of these three queries. The current generation of optimizers should be capable of recognizing that the remaining selection on TENKTUP1 can be propogated to TENKTUP2 since the selection and join predicates involve the same attribute.

The results in [BITT83] provided dramatic evidence of the relative performance of the different join algorithms. For example, in 1983, nested loops was the only join method supported by the IDM 500 and ORACLE. Each required over 5 hours to execute the joinAselB query when no indices were available. The commercial version of INGRES, on the other hand, included a sort-merge join method and could execute the same query in about 2 minutes! With indices, all the systems performed similarly. DIRECT's use of parallelism was somewhat more successful than it had been with the selection tests, providing a speedup factor of about 2.58 with 4 processors. Its performance relative to the other systems was not all that impressive and led us to conclude that "limited parallelism" and a "dumb" algorithm (parallel nested loops) could not provide the same level of performance as a "smart" algorithm (sort merge) and "no parallelism".

2.2.3. Projection Queries

Implementation of the projection operation is normally done in two phases in the general case. First a pass is made through the source relation to discard unwanted attributes. A second phase is necessary in to eliminate any duplicate tuples that may have been introduced as a side effect of the first phase (i.e. elimination of an attribute which is the key or some part of the key). The first phase requires a complete scan of the relation. The second phase is normally performed in two steps. First, the relation is sorted to bring duplicate tuples together. Next, a sequential pass is made through the sorted relation, comparing neighboring tuples to see if they are identical (some sorts have an option to eliminate duplicates). Alternatively, hashing can be used to gather duplicate tuples together. Secondary storage structures such as indices are not useful in performing this operation. In the special case where the projected fields contain a unique key, the duplicate elimination phase can be skipped. That case is not tested by the benchmark.

Initially, a wide variety of queries were considered, but we discovered that only two queries were need to obtain indicative results on the relative performance of the different systems. The first query (Query 18, Appendix I) has a projection factor of 1%, eliminating 99% of the relation as duplicates. For a 10,000 tuple relation, the result of this query contains 100 tuples. The second query (Query 19, Appendix I) has a 100% projection factor. Although this query

eliminates no duplicates, the result relation must still be sorted and then scanned for duplicates because the schema does not declare the projected fields to contain a unique key. This particular query provides an estimate of the cost of checking the result relation of an arbitrary query for duplicates.

2.2.4. Aggregate Queries

The aggregate queries in the Wisconsin benchmark consider both scalar aggregate operations (e.g. the minimum value of an attribute) and complex aggregate functions. In an aggregate function, the tuples of a relation are first partitioned into non-overlapping subsets using the SQL "group by" construct. After partitioning, an aggregate operation such as MIN is computed for each partition.

The benchmark contains three aggregate queries with two versions of each (without any indices and with a secondary, non-clustered index):

1) MIN scalar aggregate queries (Queries 20 and 23)
2) MIN aggregate function queries (Queries 21 and 24)
3) SUM aggregate function queries (Queries 22 and 25)

The motivation for including an indexed version of each of these three queries was to determine whether any of the query optimizers would use an index to reduce the execution time of the queries. For the MIN scalar aggregate query, a very smart query optimizer could recognize that the query could be executed using the index alone. While none of the query optimizers tested in [BITT83] performed this optimization, subsequent optimizers for the IDM 500 (and perhaps other systems as well) implemented this "trick" to speed the execution of such operations.

For the two aggregate function queries, we had anticipated that any attempt to use the secondary, non-clustered index on the partitioning attribute would actually slow the query down as a scan of the complete relation through such an index will generally result in each data page being accessed several times. One alternative algorithm is to ignore the index, sort on the partitioning attribute, and then make a final pass collecting the results. Another algorithm which works very well if the number of partitions is not too large is to make a single pass through the relation hashing on the partitioning attribute.

2.2.5. Update Queries

The update queries are probably the weakest aspect of the benchmark as only the following four simple update queries are included:

1) Insert 1 tuple
2) Update key attribute of 1 tuple
3) Update non-key attribute of 1 tuple
4) Delete 1 tuple

The principal objective of these queries was to examine the impact of clustered and non-clustered indices on the cost of updating, appending, or deleting a tuple. In addition, the queries indicate the advantage of having an index to help locate the tuple to be modified. To accomplish this two versions of each query were run: with and without indices (one primary-clustered index on the unique2 attribute, a unique-non-clustered index on the unique1 attribute, and a non-unique-non-clustered index on the onePercent attribute). It should be noted, however, that not enough updates were performed to cause a significant reorganization of the index pages. A more realistic evaluation of update queries would require running these queries in a multiprogramming environment, so that the effects of concurrency control and deadlocks were measured. In addition, bulk updates should have been included.

Even though these update queries are very (too) simple, they produced a number of interesting results in [BITT83]. First, was the low cost of an append compared to that of a delete, in the no-index case. The explanation for this discrepancy is that new tuples are generally inserted near the end of the file without checking if they are a duplicate of an existing tuple. Thus, appending a tuple only involves writing a new tuple. Deleting a tuple requires finding the tuple. Without an index, this requires that the entire relation be scanned. The performance of each system tested on the "modify non-key" query (i.e., modify a tuple identified by a qualification on a non-key, but indexed, attribute) demonstrated a very efficient use of a secondary index to locate the tuple. However, one could again argue that the right algorithm for this query would require verifying that the modified tuple does not introduce a duplicate tuple.

Another interesting result occurs when the update query:

```
UPDATE TENKTUP1
SET unique2=10002
WHERE unique2 = 1491
```

(Query 28 Appendix I).

is executed in the presence of a clustered index on the unique2 attribute. This query is similar to one that causes a problem known as the Halloween Problem [TAND87] to occur. For example, if the query:

```
UPDATE payroll
SET salary = salary*1.1
```

is executed using an index on the salary attribute, it will run forever if the updated tuples are inserted directly back into the index. As a simplistic solution to the problem, some systems (e.g., System R and early versions of SQL/DS) refused to use an index on any attribute if that attribute was one of the ones being updated; instead, resorting, if necessary, to a sequential scan to process the query. Query 28 is not an instance of the Halloween Problem and it is incorrect for the optimizer to treat it as such.

2.2.6. Evaluation

The Wisconsin benchmark, while certainly not perfect, did a good job of discovering performance anomalies in the early relational DBMS products. While it no longer receives much public attention, a number of vendors and users still run it as part of their standard quality assurance and performance test suites. With scaled-up relations, it is remains a fairly thorough single-user evaluation of the basic operations that a relational system must provide.

The benchmark has been criticized for a number of deficiencies [TURB87, BITT88, ONEI91]. These criticisms are certainly valid; in particular, its single-user nature, the absence of bulk updates, database load and unload tests, lack of outer join tests, and the relative simplicity of the various complex join queries. It is not so obvious that other criticisms such as its weak collection of data types and the difficulty of scaling are correct. For example, the design of the original string attributes and the lack of floating point or decimal attributes have been widely criticized. While we admit that the design of the original strings was deficient, the fact of the matter is that benchmark was designed to study the relative performance of two database systems and not the relative string and integer performance of a single system.

In particular, we assert that, if system A is 20% faster than system B processing selection predicates on integer attributes that the relative performance of the two systems would vary only slightly if floating-point or string attributes were used instead (assuming, of course, a constant hardware platform). The only possible way that changing the attribute type could change the

relative performance of two systems significantly is if the extra comparisons caused a system to change from being I/O bound to being CPU bound.

In our opinion, other than its single user nature, the second most significant problem with the benchmark today is that the join queries it contains are too simple. The benchmark should be augmented to include much more complex join queries with a wider range of join selectivity factors. In addition, a cost component should have been included so that one could meaningfully compare the response times of systems with different costs. As suggested above, such an extension is straightforward.

3.0. Benchmarking Parallel Database Systems using the Wisconsin Benchmark

While the Wisconsin benchmark is no longer widely used to evaluate relational database systems on mono-processor configurations, it has been fairly extensively used to evaluate database systems running on a parallel processors including Teradata [DEWI87], Gamma [DEWI88, DEWI90], Tandem [ENGL89a], and Volcano [GRAE90]. In this section we describe how the Wisconsin benchmark can be used to measure the speedup and scaleup characteristics of such parallel systems. This discussion is illustrated with an evaluation of the Gamma database machine [DEW90].

3.1. Speedup and Scalup: Two key metrics for a Parallel Database System

As illustrated by Figure 4, *speedup* and *scaleup* are widely accepted as the two key metrics for evaluating the performance of a parallel database system [ENGL89a, SMIT89] (see Figure 4). ***Speedup*** is an interesting metric because it indicates whether adding additional processors and disks to a system results in a corresponding decrease in the response time for a query. A system provides what is termed *linear* speedup if twice as much hardware can perform a given task in half the elapsed time. Figure 5.a illustrates the ideal speedup curve. However, simply running a standard relational database system on a multiprocessor will not necessarily exhibit any degree of speedup unless the system decomposes each of the relational operators into subunits that can be executed independently and in parallel (Figure 5.b). While techniques to do this decomposition are now fairly well understood, our evaluation of DIRECT in [BITT83] demonstrated that this was not always the case.

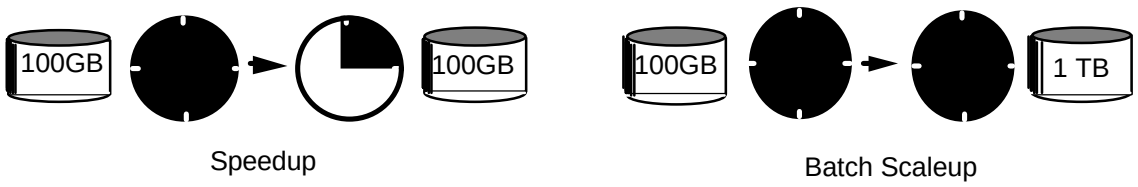

Figure 4. The difference between a speedup design in which a one-minute job is done in 15-seconds, and a scaleup design in which a ten-times bigger job is done in the same time by a ten-times bigger system.

As illustrated by Figure 5.c, the barriers to achieving linear speedup are ***startup***, ***interference,*** and ***skew. Startup*** is the time needed to start a parallel computation. If thousands of processes must be started, this can easily dominate the actual computation time. This is especially true if the startup overhead for a query is a significant fraction of the total execution time of the query, as it can be for simple queries that retrieve and process only a few tuples. ***Interference*** is the slowdown each new process imposes on the others. Even if the interference is only 1% per process, with 50 processors and one process/processor such a system will have a maximum speedup of 25. Finally, ***skew*** (variance) in the service times of the processes executing a job can begin to limit the obtainable speedup when an operation is divided into many very small pieces.

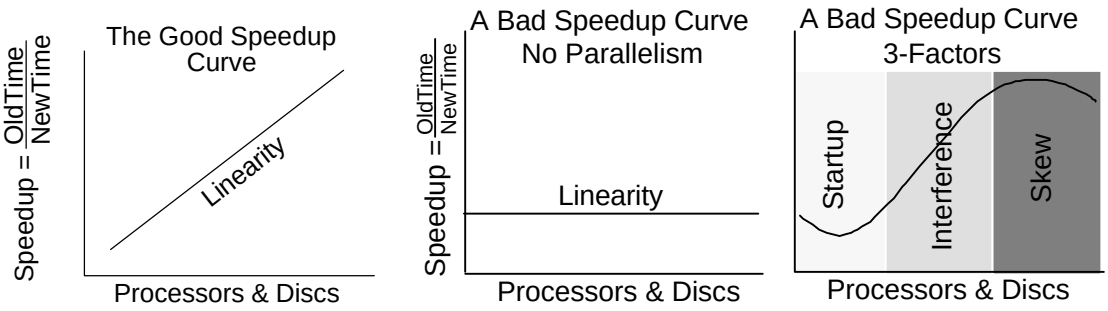

Figure 5.a **Figure 5.b** **Figure 5.c**

Figure 5: The standard speedup curves. The upper left curve is the ideal. The upper middle graph shows no speedup as hardware is added. The upper right curve shows the three threats to parallelism. Initial startup costs may dominate at first. As the number of processes increase, interference can increase. Ultimately, the job is divided so finely, that the variance in service times (skew) causes a slowdown.

The second key metric for parallel database systems is ***scaleup***. Basically scaleup measures whether a constant response time can be maintained as the workload is increased by adding a proportional numbers of processors and disks. The motivation for evaluating the extent to when a database system scales is that certain, generally batch, jobs must be must be completed in a given window of time, regardless of whether the workload increases or not. While the traditional solution has been to buy ever-faster mainframes, parallel database systems that exhibit

flat scaleup characteristics (such as those marketed by Teradata [TERA83, TERA85] and Tandem [TAND87, TAND88]) provide a way of way of incrementally responding to increasing batch workloads. The ideal batch scaleup curve is shown in Figure 6.a. A constant response time is maintained as the size of the problem and system grow incrementally. Figure 6.b illustrates a bad scaleup curve for a system with the response time growing even though resources proportional to increases in the workload are added.

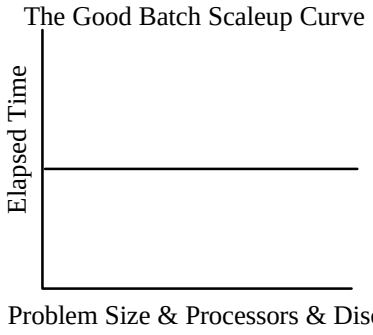

Figure 6.a

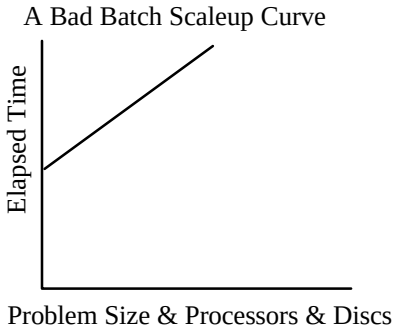

Figure 6.b

Figure 6: Batch scaleup curves: Figure 6.a. is a good batch scaleup curve showing constant processing time as proportionately more processing elements are applied to a proportionately larger problem. Figure 6.b. is a bad scaleup curve showing that as the problem grows the elapsed time grows even though more processing and disk elements are applied to the problem.

Another metric related to scaleup is ***sizeup*** in which the system configuration is kept constant while the workload is increased. A system is said to exhibit good sizeup characteristics if doubling the size of the data set being evaluated does not result in more than a two-fold increase in the response time for a query. While sizeup can be considered to be a special case of the scaleup test in which the hardware configuration is kept constant, certain relational operations such as the sorting, the sort-merge join method, and b-tree searches are logarithmic in nature. Since sorting is an NlogN operation (where N is the number of records to be sorted), one would expect to only obtain sublinear sizeup results. On the other hand, since the number of levels in a b-tree grows logarithmically, b-tree searches are likely to exhibit superlinear sizeup characteristics. For example, assume that a b-tree on the unique2 attribute of a 100,000 tuple relation is 2 levels deep and that the corresponding index on a one million tuple relation is 3 levels deep. Assuming no buffer pool hits, a single tuple indexed selection of the 100,000 tuple relation will require 3 I/Os (two index I/Os plus one I/O to fetch the appropriate data page). The same query on the million tuple relation will require 4, and not 30, I/Os. While sizeup is an interesting metric in itself, the real value of conducting sizeup measurements on a parallel database system is as an aid in understanding non-linear scaleup results.

3.2. Using the Wisconsin Benchmark to Measure Speedup, Scaleup, and Sizeup

By their definition speedup tests are conducted by fixing the size of the relations in the database and the suite of queries while the number of processors and disks is varied. There are, however, several issues that must be carefully considered. First, the size of the benchmark relations must be chosen carefully. They must be large enough so that the number of levels in each index remains constant as the number of disks over which the relation has been declustered is increased to avoid artificially producing superlinear speedup results. The actual size will depend on a number factors including the underlying page size that is used for the indices and the maximum number of disks over which a relation is declustered. For example, when a 1 million tuple relation is stored on a single disk, the indices on the unique1 and unique2 attributes will each contain 1 million entries and will be three levels deep (assuming that index entries consist of a 4 byte key field, a 4 byte count field, and an 8 byte record_id field). If the same relation is declustered over 100 disks, each disk will contain only 100,000 tuples and the corresponding indices will only be 2 levels deep. As discussed above, such changes tend to artificially produce superlinear speedup results and should be avoided, if possible. On the other hand, the relation cannot be so large that it will not fit on a single disk drive if a single processor/disk pair is to be used as the base case. In some cases it may be impossible to satisfy both these constraints. In this case, the solution is use more than a single processor/disk as the base case.

The second issue that must be considered when setting up a database for measuring the speedup of a system is interprocessor communication. If a single processor is used as the base case (assuming that the base relations fit) no interprocessor communications will occur. As the system is scaled from one to two processors the interprocessor communications traffic may increase substantially - depending on the actual query being executed and the way in which the result relation is declustered. As a consequence, the observed speedups may very well be sublinear. While this is reasonable if one is interested is determining the absolute speedup achievable by a system, in some circumstances the objective is to predict what would happen if a currently operational system were scaled from 10 to 100 processors. Thus, for some applications it may be valid to use more than a single processor as the base case.

Designing scaleup experiments is somewhat simpler. Again, the same query suite is used with each configuration tested. The basic decisions that must be made are to select a base relation size and a base hardware configuration which is scaled proportionally to increases in the sizes of the base relations. For example, assume that one million tuple benchmark relations are selected along with a base hardware configuration of 5 processors, each with one disk. If the benchmark

relations are doubled in size, the hardware configuration is also doubled; expanding to 10 processors with disks. As with the speedup tests, the selection of a base configuration size can have a significant impact on the results obtained. Typically a base configuration of between 2 and 5 processors is a reasonable compromise.

3.3. Sizeup, Speedup, and Scaleup Experiments on the Gamma Prototype

This section describes the results of conducting sizeup, speedup, and scaleup experiments on the Gamma prototype using a subset of the selection and join queries from the Wisconsin benchmark. It begins with an overview of the Gamma database machine. This is followed by a set of sizeup, speedup, and scaleup experiments on a subset of the selection and join queries from the Wisconsin benchmark. A more complete set of tests can be found in [DEWI90]. A similar set of tests on Release 2 of Tandem's NonStop SQL system is described in [ENGL89a].

The tests began by constructing 100,000, 1 million, and 10 million tuple versions of the benchmark relations. Two copies of each relation were created and loaded. Except where noted otherwise, tuples were declustered by hash partitioning on the unique1 attribute. In all cases, the results presented represent the average response time of a number of equivalent queries. Gamma was configured to use a disk page size of 8K bytes and a buffer pool of 2 megabytes per processor. The results of all queries were stored in the database. Returning data to the host was avoided in order to mask the speed of the communications link between the host and the database machine and the speed of host processor itself. By storing the result relations in the database, the impact of these factors was minimized - at the expense of incurring the cost of declustering and storing the result relations.

3.3.1 Overview of the Gamma Database Machine

Hardware

The design of the Gamma database machine is based on a shared-nothing architecture [STON86], in which processors do not share disk drives or random access memory. They can only communicate with one another by sending messages through an interconnection network. Mass storage in such an architecture is distributed among the processors by connecting one or more disk drives to each processor as shown in Figure 7. This architecture characterizes the database systems being used by Teradata [TERA85], Gamma [DEWI86, DEWI90], Tandem [TAND88], Bubba [ALEX88, COPE88], and Arbre [LORI89].

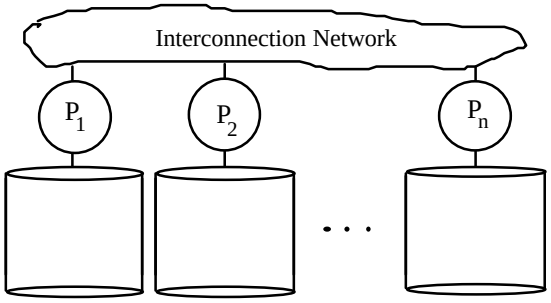

Figure 7. The basic shared nothing design. Each processor has a private memory and one or more disk drives. Processors communicate via a high-speed interconnect network.

The hardware platform currently used by Gamma is a 32 processor Intel iPSC/2 hypercube. Each processor is configured with a Intel 386 CPU, 8 megabytes of memory, and a 330-megabyte MAXTOR 4380 (5 1/4") disk drive. Each disk drive has an embedded SCSI controller which provides a 45 Kbyte RAM buffer that acts as a disk cache for sequential read operations. The nodes in the hypercube are interconnected to form a hypercube using custom VLSI routing modules. Each module supports eight[2] full-duplex, serial, reliable communication channels each operating at 2.8 megabytes/second. Small messages (<= 100 bytes) are sent as datagrams and take less than a millisecond. For large messages, the hardware builds a communications circuit between the two nodes over which the entire message is transmitted without software overhead or copying. After the message has been completely transmitted, the circuit is released. As an example of the performance obtainable when sending page-sized messages, an 8 Kbyte message takes about 4.5 milliseconds.

Physical Database Organization

Relations in Gamma are **declustered** [LIVN87] (see Figure 8) across all disk drives in the system.[3] Declustering a relation involves distributing the tuples of a relation among two or more disk drives according to some distribution criteria such as applying a hash function to the key attribute of each tuple. One of the key reasons for using declustering in a parallel database system is to enable the DBMS software to exploit the I/O bandwidth by reading and writing multiple disks in parallel. By declustering the tuples of a relation the task of parallelizing a scan operator becomes trivial. All that is required is to start a copy of the operator on each processor or disk containing relevant tuples, and to merge their outputs at the destination.

---

[2] On configurations with a mix of compute and I/O nodes, one of the 8 channels is dedicated for communication to the I/O subsystem.

[3] Declustering has its origins in the concept of horizontal partitioning initially developed as a distribution mechanism for distributed DBMS [RIES78].

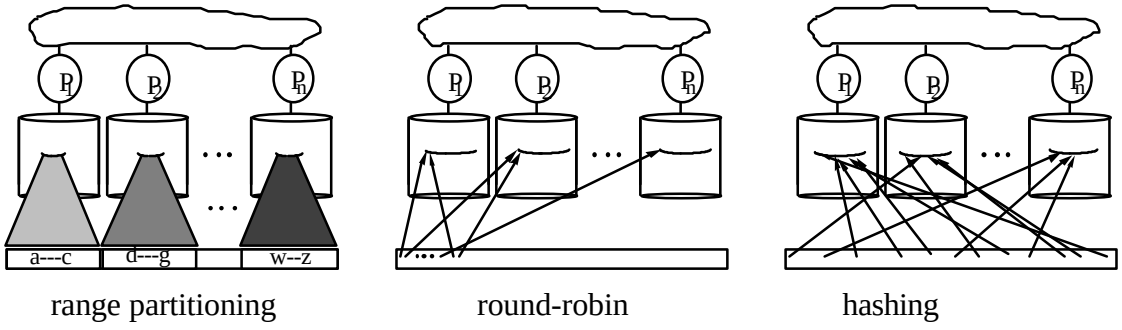

Figure 8: The three basic declustering schemes: range declustering maps contiguous fragments of a table to various disks. Round-Robin declustering maps the i'th record to disk i **mod** n. Hashed declustering, maps each record to a disk location based on some hash function. Each of these schemes spreads data among a collection of disks, allowing parallel disk access and parallel processing.

Gamma currently provides the user with three alternative declustering strategies: **round robin**, **hashed,** and **range partitioned**. With the first strategy, tuples are distributed in a round-robin fashion among the disk drives. This is the default strategy and is used for all relations created as the result of a query. If the hashed partitioning strategy is selected, a randomizing function is applied to the key attribute of each tuple to select a storage unit. In the third strategy the user specifies a range of key values for each node. The partitioning information for each relation is stored in the database catalog. Once a relation has been partitioned, Gamma provides the normal collection of access methods including both clustered and non-clustered indices. When the user requests that an index be created on a relation, the system automatically creates an index on each fragment of the relation. Gamma does not require that the clustered index for a relation be constructed on the partitioning attribute.

As a query is being optimized, the partitioning information for each source relation in the query is incorporated into the query plan produced by the query optimizer. In the case of hash and range-partitioned relations, this partitioning information is used by the query scheduler (discussed below) to restrict the number of processors involved in the execution of selection queries on the partitioning attribute. For example, if relation X is hash partitioned on attribute y, it is possible to direct selection operations with predicates of the form "X.y = Constant" to a single node; avoiding the use of any other nodes in the execution of the query. In the case of range-partitioned relations, the query scheduler can restrict the execution of the query to only those processors whose ranges overlap the range of the selection predicate (which may be either an equality or range predicate).

Software Architecture

Gamma is built on top of an operating system designed specifically for supporting database management systems. NOSE provides multiple, lightweight processes with shared memory. A non-preemptive scheduling policy is used to help prevent convoys [BLAS79] from occurring. NOSE provides communications between processes using the reliable message passing hardware of the Intel iPSC/2 hypercube. File services in NOSE are based on the Wisconsin Storage System (WiSS) [CHOU85].

The algorithms for all the relational operators are written as if they were to be run on a single processor. As shown in Figure 9, the input to an Operator Process is a stream of tuples and the output is a stream of tuples that is demultiplexed through a structure we term a **split table**. Once the process begins execution, it continuously reads tuples from its input stream (which may be the output of another process or a table), operates on each tuple, and uses a split table to route the resulting tuple to the process indicated in the split table.

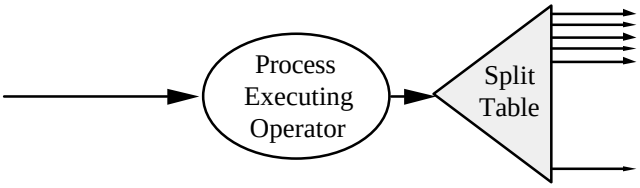

Figure 9: A relational dataflow graph showing a relational operator's output being decomposed by a split table into several independent streams. Each stream may be a duplicate of the input stream, or a partitioning of the input stream into many disjoint streams. With the split and merge operators, a web of simple sequential dataflow nodes can be connected to form a parallel execution plan.

As an example, consider the query shown in Figure 10 in conjunction with the split tables in Figure 11. Assume that three processes are used to execute the join operator, and that five other processes execute the two scan operators - three processes are used to scan the 3 partitions of table A and two processes are used to scan the 2 partitions of table B. Each of the three table A scan nodes will have the same split table, sending all tuples with join attributes values between "A-H" to port 1 of join process 0, all between "I-Q" to port 1 of join process 1, and all between "R-Z" to port 1 of join process 2. Similarly the two table B scan nodes have the same split table except that their outputs are merged by port 1 (not port 0) of each join process. Each join process sees a sequential input stream of A tuples from the port 0 merge (the left scan nodes) and another sequential stream of B tuples from the port 1 merge (the right scan nodes). Four join methods are available to the query optimizer to choose from [DEWI84, SCHN89]: simple hash,

grace hash, hybrid hash, and sort-merge. Additional details on how queries are processed by Gamma can be found in [DEWI90].

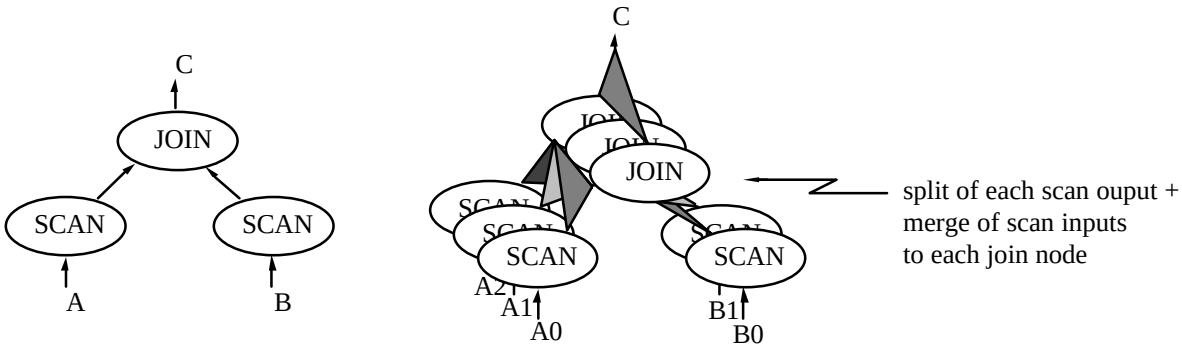

Figure 10: A simple relational dataflow graph showing two relational scans (project and select) consuming two input tables, A and B and feeding their outputs to a join operator which in turn produces a data stream C.

Table A Scan Split Table		Table B Scan Split Table	
Predicate	Destination Process	Predicate	Destination Process
"A-H"	(cpu #5, Process #3, Port #0)	"A-H"	(cpu #5, Process #3, Port #1)
"I-Q"	(cpu #7, Process #8, Port #0)	"I-Q"	(cpu #7, Process #8, Port #1)
"R-Z"	(cpu #2, Process #2, Port #0)	"R-Z"	(cpu #2, Process #2, Port #1)

Figure 11. Sample Split Tables which map tuples to different output streams (ports of other processes) depending on the range value of some attribute of the input tuple.The split table on the left is for the Table A scan in figure 7, while the table on the right is for the table B scan.

3.3.1. Sizeup Experiments

Selection Queries

The first set of selection tests were designed to determine how Gamma would respond as the size of the source relations was increased while the machine configuration was kept at 30 processors with disks. Ideally, the response time of a query should grow as a linear function of the size of input and result relations. For these tests six different selection queries (queries 1 to 5 and 7 from Appendix I) were run on three sets of relations containing, respectively, 0.1 million, 1 million, and 10 million tuples. Query 6, a 10% selection through a non-clustered index query, is not included as the Gamma query optimizer chooses to use a sequential scan for this query. Except for query 7, the predicate of each query specifies a range of values and, thus, since the input relations were declustered by hashing, the query must be sent to all the nodes.

The results from these tests are tabulated in Table 3. For the most part, the execution time for each query scales as a fairly linear function of the size of the input and output relations. There are, however, several cases where the scaling is not perfectly linear. Consider, first the 1% non-indexed selection. As the size of the input relation is increased from 1 to 10 million tuples, the response time increase is almost perfectly linear (8.16 secs. to 81.15 secs.). But, the response time increase from 100,000 tuples to 1 million tuples (0.45 sec. to 8.16 sec) is sub-linear (an 18-fold increase in response time for a 10-fold increase in database size giving a .55 sizeup benefit). The cause of this sublinearity is that the cost of starting a query on 30 processors (approximately 0.24 seconds) constitutes almost 1/2 the total execution time of the selection on the 100,000 tuple relation. The 10% selection using a clustered index is another example where increasing the size of the input relation by a factor of ten results in more than a ten-fold increase in the response time for the query. This query takes 5.02 seconds on the 1 million tuple relation and 61.86 seconds on the 10 million tuple relation. To understand why this happens one must consider the impact of seek time on the execution time of the query. Since two copies of each relation were loaded, when two one million tuple relations are declustered over 30 disk drives, the fragments occupy approximately 53 cylinders (out of 1224) on each disk drive. Two ten million tuple relations fill about 530 cylinders on each drive. As each page of the result relation is written to disk, the disk heads must be moved from their current position over the input relation to a free block on the disk. Thus, with the 10 million tuple relation, the cost of writing each output page is much higher. This effect does not occur with the 1% non-indexed selection since the result relation fits in the buffer pool and can be written to disk sequentially at the end of the query.

Table 3 - Selection Queries
30 Processors With Disks
(All Execution Times in Seconds)

	Number of Tuples in Source Relation		
Query Description	**100,000**	**1,000,000**	**10,000,000**
1% non-indexed selection	0.45	8.16	81.15
10% non-indexed selection	0.82	10.82	135.61
1% selection using clustered index	0.35	0.82	5.12
10% selection using clustered index	0.77	5.02	61.86
1% selection using non-clustered index	0.60	8.77	113.37
single tuple select using clustered index	0.08	0.08	0.14

As expected, the use of a clustered B-tree index always provides a significant improvement in performance. One observation to be made from Table 3 is the relative consistency of the execution time of the selection queries through a clustered index. Notice that the execution

time for a 10% selection on the 1 million tuple relation is almost identical to the execution time of the 1% selection on the 10 million tuple relation. In both cases, 100,000 tuples are retrieved and stored, resulting in identical I/O and CPU costs.

The final row of Table 3 presents the time required to select a single tuple using a clustered index and return it to the host. Since the selection predicate is on the partitioning attribute, the query is directed to a single node, avoiding the overhead of starting the query on all 30 processors. The response for this query increases significantly as the relation size is increased from 1 million to 10 million tuples because the height of the B-tree increases from two to three levels. It is interesting to note how the response time increases as the size of the relation increases. While the B-tree for all three relations is only two levels deep (at each node), the number of entries in the root node increases from 4 to 330 as the relation size is increased from 100,000 to 10 million tuples. Consequently, significantly more time is required to do a binary search of the root node to locate the correct leaf page.

Join Queries

Two join queries were used for the join scaleup tests: joinABprime and joinAselB. The A and B relations contain either 0.1 million, 1 million, or 10 million tuples. The Bprime relation contains, respectively, 10,000, 100,000, or 1 million tuples.

The first variation of the join queries tested involved no indices and used a non-partitioning attribute for both the join and selection attributes. Thus, before the join can be performed, the two input relations must be redistributed by hashing on the join attribute value of each tuple. The results from these tests are contained in the first 2 rows of Table 4. The second variation of the join queries also did not employ any indices but, in this case, the relations were hash partitioned on the joining attribute; enabling the redistribution phase of the join to be skipped. The results for these tests are contained in the last 2 rows of Table 4.

Table 4 - Join Queries
30 Processors With Disks
(All Execution Times in Seconds)

Query Description	Number of Tuples in Relation A 100,000	1,000,000	10,000,000
JoinABprime with non-partitioning attributes of A and B used as join attributes	3.52	28.69	438.90
JoinAselB with non-partitioning attributes of A and B used as join attributes	2.69	25.13	373.98
JoinABprime with partitioning attributes of A and B used as join attributes	3.34	25.95	426.25
JoinAselB with partitioning attributes of A and B used as join attributes	2.74	23.77	362.89

The results in Table 4 indicate that the execution time of each join query increases in a fairly linear fashion as the size of the input relations are increased. Gamma does not exhibit linearity for the 10 million tuple queries because the size of the inner relation (208 megabytes) is twice as large as the total available space for hash tables. Hence, the Hybrid join algorithm [SCHN89] needs two buckets to process these queries. While the tuples in the first bucket can be placed directly into memory-resident hash tables, the second bucket must be written to disk.

As expected, the version of each query in which the partitioning attribute was used as the join attribute ran faster. From these results one can estimate a lower bound on the aggregate rate at which data can be redistributed by the Intel iPSC/2 hypercube. Consider the version of the joinABprime query in which a million tuple relation is joined with a 100,000 tuple relation. This query requires 28.69 seconds when the join is not on the partitioning attribute. During the execution of this query, 1.1 million 208 byte tuples must be redistributed by hashing on the join attribute, yielding an aggregate total transfer rate of 7.9 megabytes/second during the processing of this query. This should not be construed, however, as an accurate estimate of the maximum obtainable interprocessor communications bandwidth as the CPUs may be the limiting factor (the disks are not likely to be the limiting factor as from Table 3 one can estimate that the aggregate bandwidth of the 30 disks to be about 25 megabytes/second).

3.3.2 Speedup Experiments

Selection Queries

This section examines how the response times for both the non-indexed and indexed selection queries on the 1 million tuple relation are affected by the number of processors used to execute the query.[4] Ideally, one would like to see a linear improvement in performance as the number of processors is increased from 1 to 30. Increasing the number of processors increases both the aggregate CPU power and I/O bandwidth available, while reducing the number of tuples that must be processed by each processor.

Figure 12 presents the average response times for the non-indexed 1% and 10% selection queries (queries 1 and 2) on the one million tuple relation. As expected, the response time for each query decreases as the number of processors and disks is increased. The response time is higher for the 10% selection due to the cost of declustering and storing the result relation. While one could always store result tuples locally, by partitioning all result relations in a round-robin (or hashed) fashion one can ensure that the fragments of every result relation each contain approximately the same number of tuples.

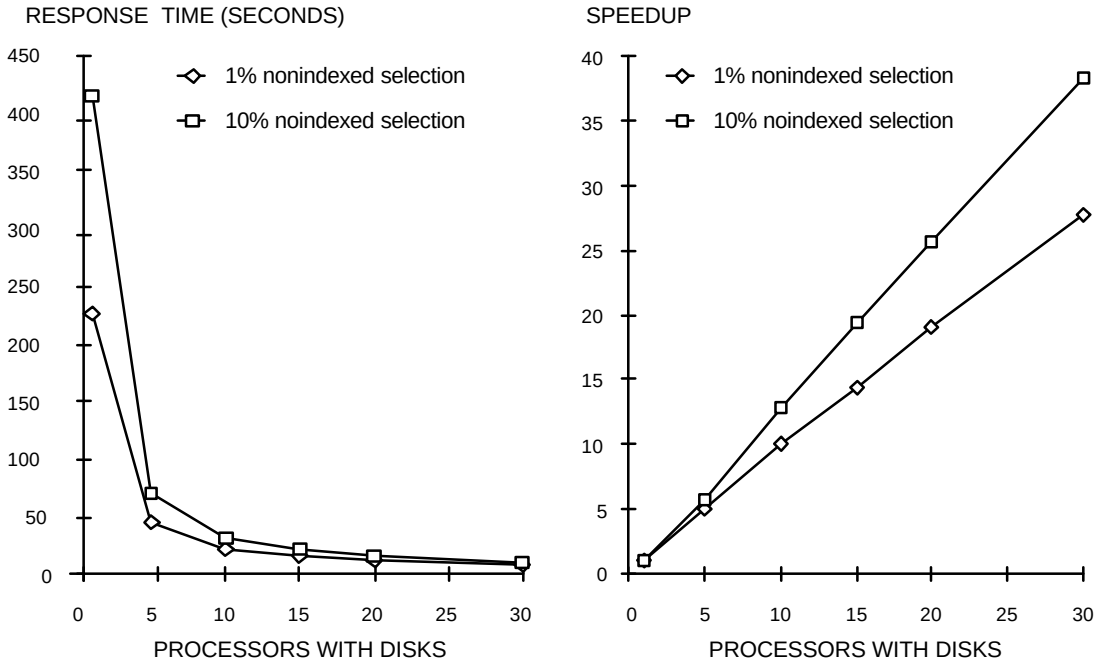

Figure 12: Response times and speedup of non-indexed selection queries.

[4] The 1 million tuple relation was used for these experiments because the 10 million tuple relation would not fit on 1 disk drive.

Figure 13 presents the average response time and speedup as a function of the number of processors for the following three queries: a 1% selection through a clustered index (query 3), a 10% selection through a clustered index (query 4), and a 1% selection through a non-clustered index (query 5), all accessing the 1 million tuple relation.

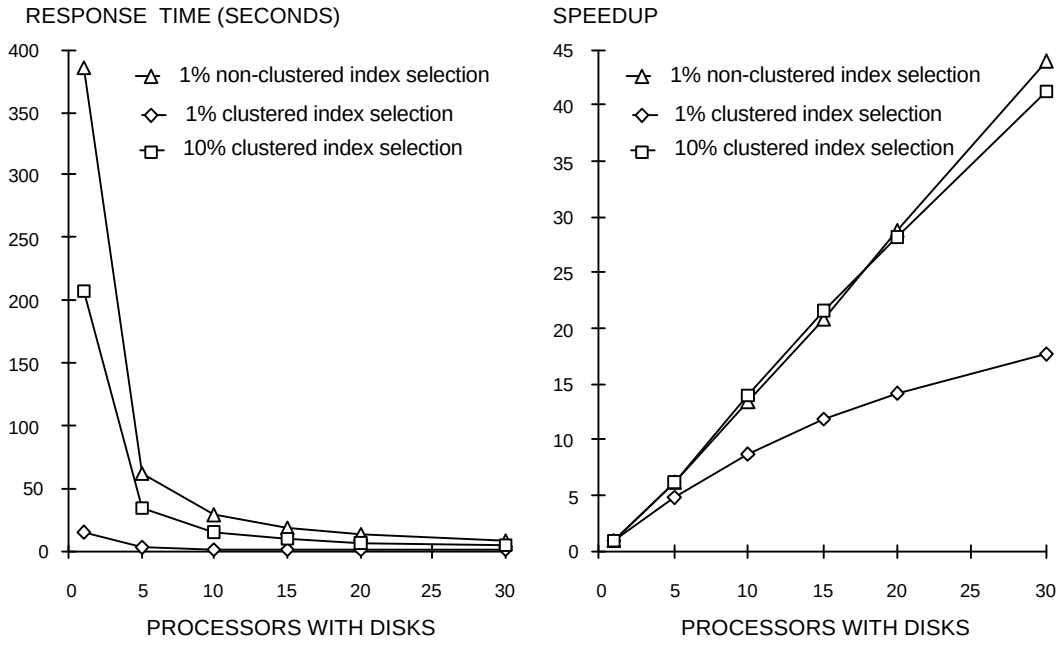

Figure 13: Response times and speedup of indexed selection queries.

Three queries in Figures 12 and 13 have super-linear speedup, one is slightly sublinear, and one is significantly sublinear. Consider first the 10% selection via a relation scan, the 1% selection through a non-clustered index, and the 10% selection through a clustered index. As discussed above, the source of the superlinear speedups exhibited by these queries is due to significant differences in the time the various configurations spend seeking. With one processor, the 1 million tuple relation occupies approximately 66% of the disk. When the same relation is declustered over 30 disk drives, it occupies about 2% of each disk. In the case of the 1% non-clustered index selection, each tuple selected requires a random seek. With one processor, the range of the each random seek is approximately 800 cylinders while with 30 processors the range of the seek is limited to about 27 cylinders. Since the seek time is proportional to the square root of the distance traveled by the disk head [PATT89], reducing the size of the relation fragment on each disk significantly reduces the amount of time that the query spends seeking.

A similar effect occurs with the 10% clustered index selection. In this case, once the index has been used to locate the tuples satisfying the query, each input page will produce one output

page and at some point the buffer pool will be filled with dirty output pages. In order to write an output page, the disk head must be moved from its position over the input relation to the position on the disk where the output pages are to be placed. The relative cost of this seek decreases proportionally as the number of processors increases, resulting in a superlinear speedup for the query. The 10% non-indexed selection shown in Figure 13 is also superlinear for similar reasons. The reason that this query is not affected to the same degree is that, without an index, the seek time is a smaller fraction of the overall execution time of the query.

The 1% selection through a clustered index exhibits sublinear speedups because the cost of initiating a select and store operator on each processor (a total of 0.24 seconds for 30 processors) becomes a significant fraction of the total execution as the number of processors is increased.

Join Queries

For the join speedup experiments, we used the joinABprime query with a 1 million tuple A relation and a 100,000 tuple Bprime relation. Two different cases were considered. In the first case, the input relations were declustered by hashing on the join attribute. In the second case, the input relations were declustered using a different attribute. The hybrid hash-join algorithm was used for all queries. The number of processors was varied from five to thirty. Since with fewer than five processors two or more hash join buckets are needed, including the execution time for one processor (which needs 5 buckets) would have made the response times for five or more processors appear artificially fast; resulting in superlinear speedup curves.

The resulting response times and speedups are plotted in Figure 14. From the shape of these graphs it is obvious that the execution time for the query is significantly improved as additional processors are employed. Several factors prevent the system from achieving perfectly linear speedups. First, the cost of starting four operator processes (two scans, one join, and one store) on each processor increases as a function of the number of processors used. Second, the effect of short-circuiting local messages diminishes as the number of processors is increased. For example, consider a five processor configuration and the non-partitioning attribute version of the JoinABprime query. As each processor repartitions tuples by hashing on the join attribute, 1/5th of the input tuples it processes are destined for itself and will be short-circuited by the communications software. In addition, as the query produces tuples of the result relation (which is partitioned in a round-robin manner), they too will be short circuited. As the number of processors is increased, the number of short-circuited packets decreases to the point where, with

30 processors, only 1/30th of the packets will be short-circuited. Because these intra-node packets are less expensive than their corresponding inter-node packets, smaller configurations will benefit more from short-circuiting. In the case of a partitioning-attribute joins, **all** input tuples will short-circuit the network along with a fraction of the output tuples.

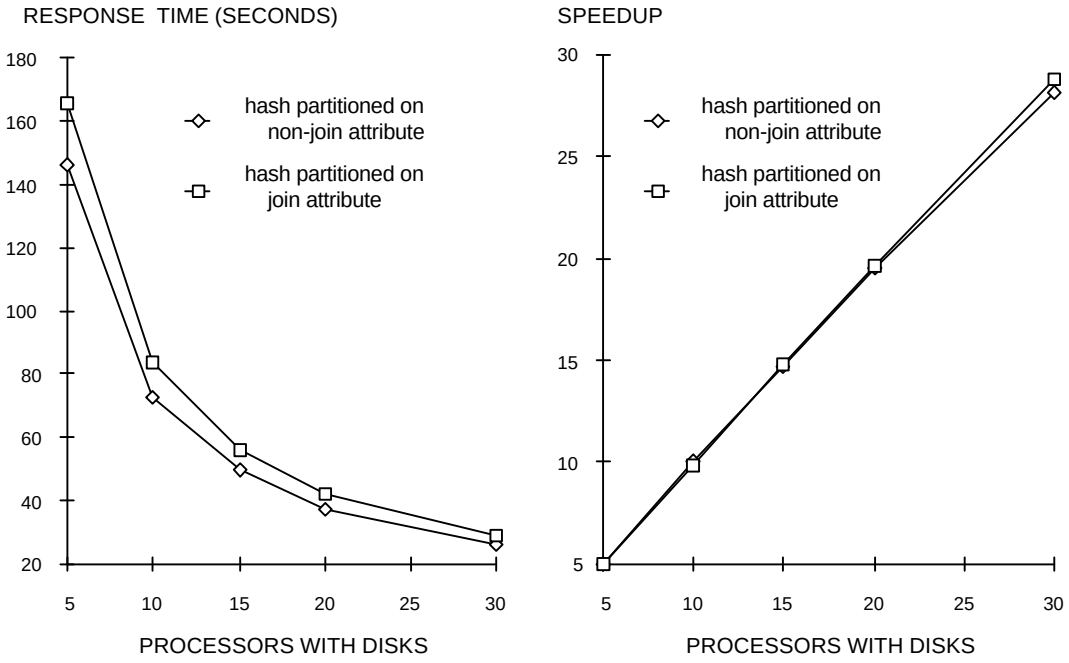

Figure 14: Response times and speedup of join queries.

3.3.3 Scaleup Experiments

Selection Queries

In the final set of selection experiments the number of processors was varied from 5 to 30 while the size of the input relations was increased from 1 million to 6 million tuples, respectively. As shown in Figure 15, the response time for each of the five selection queries remains almost constant. The slight increase in response time is due to the overhead of initiating a selection and store operator at each node. Since a single process is used to initiate the execution of a query, as the number of processors employed is increased, the load on this process is increased proportionally. Switching to a tree-based, query initiation scheme [GERB87] would distribute this overhead among all the processors.

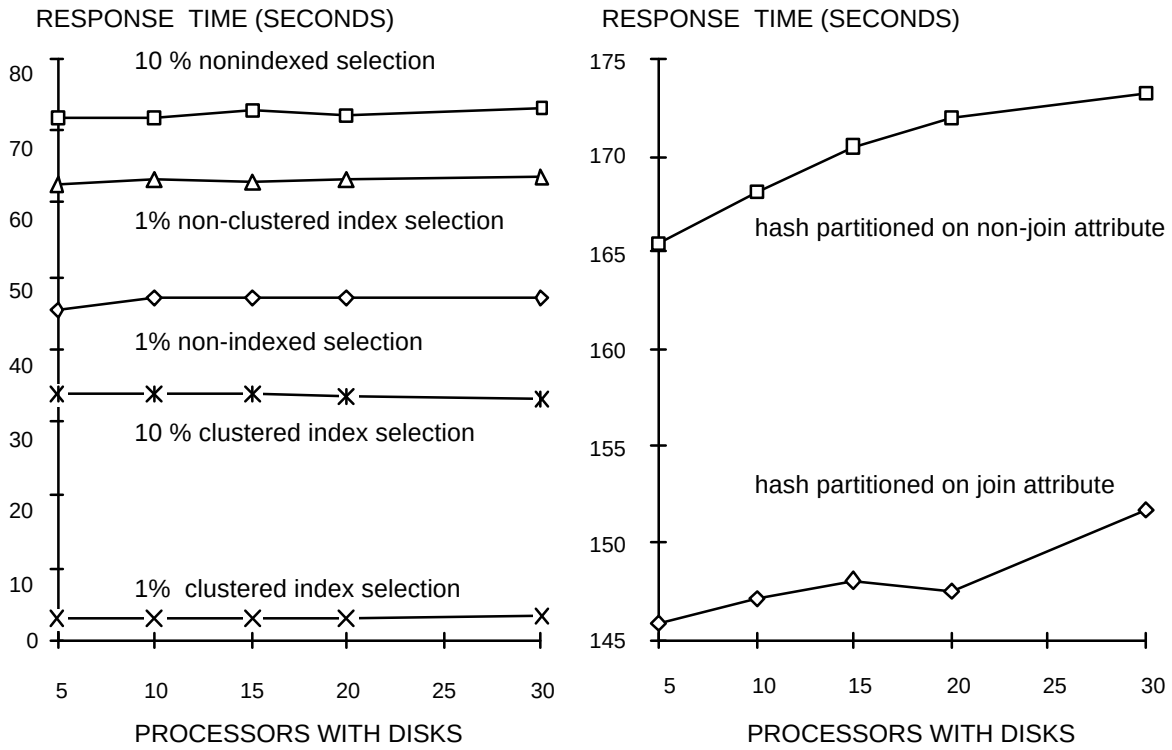

Figure 15: Scaleup response times of various queries.

Join Queries

The JoinABprime query was used for the join scaleup experiments. For these tests, the number of processors was varied from 5 to 30 while the size of the A relation was varied from 1 million to 6 million tuples in increments of 1 million tuples and the size of Bprime relation was varied from 100,000 to 600,000 tuples in increments of 100,000. Two different cases were considered. In the first case, the input relations were declustered by hashing on the join attribute. In the second case, the input relations were declustered using a different attribute. For each configuration, only one join bucket was needed. The results of these tests are presented in Figure 15. Three factors contribute to the slight increase in response times. First, the task of initiating 4 processes at each node is performed by a single processor. Second, as the number of processors increases, the effects of short-circuiting messages during the execution of these queries diminishes - especially in the case when the join attribute is not the partitioning attribute. Finally, the response time may be being limited by the speed of the communications network.

4.0. Conclusions

The original intent of the Wisconsin benchmark was to develop a relatively simple, but fairly scientific benchmark, that could be used to evaluate the performance of relational database systems and machines. We never envisioned that the benchmark would become widely used or cause the controversies that it did. In retrospect, had we realized what was to come, we would have tried to do a better job in a number of areas including more complex queries and a better suite of update queries. In the conclusions of [BITT83], we ourselves admitted that the benchmark was "neither an exhaustive comparison of the different systems, nor a realistic approximation of what measurements in a multiuser environment would be like." However, starting with a single-user benchmark provided us with insights that would have been impossible to achieve had we begun instead with multiuser experiments.

The Wisconsin benchmark did a very good job of uncovering performance and implementation flaws in the original relational products. Since then, however, these flaws have been fixed and the single-user performance differences among the various products have narrowed significantly. For the most part, the Wisconsin benchmark has been supplanted by the Debit Credit benchmark and its successor the TPC BM A benchmark, regardless of the many deficiencies of this benchmark.

Over the past few years parallel database systems have evolved from research curiosities to a highly successful products. For example, both Teradata and Tandem have each shipped systems with over two hundred processors. While mainframe vendors have found it increasingly difficult to build machines powerful enough to meet the CPU and I/O demands of relational DBMSs serving large numbers of simultaneous users or searching terabytes of data, parallel database systems based on a shared-nothing architecture [STON86] can be expanded incrementally and can be scaled to configurations containing certainly 100s and probably 1000s of processors and disks. While the DebitCredit benchmark is used as one metric for such systems, many customers of such systems use them for processing complex relational queries. As illustrated by the previous section, the Wisconsin benchmark provides a mechanism by which both the absolute performance and the speedup and scaleup characteristics of such systems can be measured.

Acknowledgements

Dina Bitton and Carolyn Turbyfill were instrumental in the development of the Wisconsin benchmark and in conducting the early tests. All too often the Wisconsin benchmark is cited as the "DeWitt" benchmark. Doing so slights the important contributions that Dina and Carolyn made in its development. Shahram Ghandeharizadeh and Donovan Schneider deserve all the credit for not only gathering the results on Gamma that were presented in Section 3, but also, their major contributions implementing Gamma itself. Finally, I would like to acknowledge Susan Englert and Jim Gray for the benchmark generator described in Section 3.

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grant DCR-8512862, by a DARPA/NASA sponsored Graduate Research Assistantship in Parallel Processing, and by research grants from Intel Scientific Computers, Tandem Computers, and Digital Equipment Corporation.

References

[ALEX88] Alexander, W., et. al., "Process and Dataflow Control in Distributed Data-Intensive Systems," Proc. ACM SIGMOD Conf., Chicago, IL, June 1988. October, 1983.

[ANON88] Anon et. al., "A Measure of Transaction Processing Power," in "Readings in Database Systems", edited by Michael Stonebraker, Morgan Kaufman, 1988.

[BLAS79] Blasgen, M. W., Gray, J., Mitoma, M., and T. Price, "The Convoy Phenomenon," Operating System Review, Vol. 13, No. 2, April, 1979.

[BITT83] Bitton, D., DeWitt, D. J., and C. Turbyfil, "Benchmarking Database Systems: A Systematic Approach," Proceedings of the 1983 Very Large Database Conference, October, 1983.

[BITT85] Bitton, D. and C. Turbyfill, "Design and Analysis of Multiuser Benchmarks for Database Systems," Proceedings of the HICSS-18 Conference, January, 1985.

[BITT88] Bitton, D. and C. Turbyfill, "A Retrospective on the Wisconsin Benchmark," in "Readings in Database Systems", edited by Michael Stonebraker, Morgan Kaufman, 1988.

[BORA82] Boral, H., DeWitt, D.J., Friedland, D., Jarrell, N., and W. K. Wilkinson, "Implementation of the Database Machine DIRECT," IEEE Transactions on Software Engineering, November, 1982.

[BORA83] Boral H. and D. J. DeWitt, "Database Machines: An Idea Whose Time has Passed? A Critique of the Future of Database Machines," Proceedings of the 3rd International Workshop on Database Machines, Munich, Germany, September, 1983.

[CHOU85] Chou, H-T, DeWitt, D. J., Katz, R., and T. Klug, "Design and Implementation of the Wisconsin Storage System (WiSS)", Software Practices and Experience, Vol. 15, No. 10, October, 1985.

[COPE88] Copeland, G., Alexander, W., Boughter, E., and T. Keller, "Data Placement in Bubba," Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.

[DEWI79] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, June, 1979.

[DEWI86] DeWitt, D., et. al., "GAMMA - A High Performance Dataflow Database Machine," Proceedings of the 1986 VLDB Conference, Japan, August 1986.

[DEWI87] DeWitt, D., Smith, M., and H. Boral, .q "A Single-User Performance Evaluation of the Teradata Database Machine," MCC Technical Report Number DB-081-87, March 5, 1987.

[DEWI88] DeWitt, D., Ghandeharizadeh, S., and D. Schneider, "A Performance Analysis of the Gamma Database Machine," Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.

[DEWI90] DeWitt, D., et. al., "The Gamma Database Machine Project," IEEE Knowledge and Data Engineering, Vol. 2, No. 1, March, 1990.

[ENGL89a] Englert, S, J. Gray, T. Kocher, and P. Shah, "A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases," Tandem Computers, Technical Report 89.4, Tandem Part No. 27469, May 1989.

[ENGL89b] Englert, S. and J. Gray, "Generating Dense-Unique Random Numbers for Synthetic Database Loading," Tandem Computers, January, 1989.

[EPST87] Epstein, R. "Today's Technology is producing High-Performance Relational Database Systems," Sybase Newsletter, 1987.

[GERB87] Gerber, R. and D. DeWitt, "The Impact of Hardware and Software Alternatives on the Performance of the Gamma Database Machine", Computer Sciences Technical Report #708, University of Wisconsin-Madison, July, 1987.

[GRAE89] Graefe, G., and K. Ward, "Dynamic Query Evaluation Plans", Proceedings of the 1989 SIGMOD Conference, Portland, OR, June 1989.

[GRAE90] Graefe, G., "Encapsulation of Parallelism in the Volcano Query Processing System," Proceedings of the 1990 ACM-SIGMOD International Conference on Management of Data, May 1990.

[GRAY88] Gray, J., H. Sammer, and S. Whitford, "Shortest Seek vs Shortest Service Time Scheduling of Mirrored Disks," Tandem Computers, December 1988.

[LIVN87] Livny, M., S. Khoshafian, and H. Boral, "Multi-Disk Management Algorithms", Proceedings of the 1987 SIGMETRICS Conference, Banff, Alberta, Canada, May, 1987.

[LORI89] Lorie, R., J. Daudenarde, G. Hallmark, J. Stamos, and H. Young, "Adding Intra-Transaction Parallelism to an Existing DBMS: Early Experience", IEEE Data Engineering Newsletter, Vol. 12, No. 1, March 1989.

[SCHN89] Schneider, D. and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", Proceedings of the 1989 SIGMOD Conference, Portland, OR, June 1989.

[ONEIL91] O'Neil, P.E. "The Set Query Benchmark", in *Database and Transaction Processing Performance Handbook*, J. Gray ed. Morgan Kauffman 1991..

[PATT89] Patterson, D. A., J. L. Hennesssy, *Computer Architecture, a Quantitative Approach*, Morgan Kaufmann, 1990.

[SCHN90] Schneider, D. and D. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," Proceedings of the Sixteenth International Conference on Very Large Data Bases", Melbourne, Australia, August, 1990.

[SMIT89] Smith, M. et. al, "An Experiment in Response Time Scalability," Proceedings of the 6th International Workshop on Database Machines," June, 1989.

[STON86] Stonebraker, M., "The Case for Shared Nothing," Database Engineering, Vol. 9, No. 1, 1986.

[TAND87] Tandem Database Group, "NonStop SQL, A Distributed, High-Performance, High-Reliability Implementation of SQL," *High Performance Transaction Systems*, Springer Verlag Lecture Notes in Computer Science 359, 1989.

[TAND88] Tandem Performance Group, "A Benchmark of Non-Stop SQL on the Debit Credit Transaction," Proceedings of the 1988 SIGMOD Conference, Chicago, IL, June 1988.

[TERA83] Teradata: DBC/1012 Data Base Computer Concepts & Facilities, Teradata Corp. Document No. C02-0001-00, 1983.

[TERA85] Teradata, "DBC/1012 Database Computer System Manual Release 2.0," Document No. C10-0001-02, Teradata Corp., NOV 1985.

Appendix I
Wisconsin Benchmark Queries for 10,000 Tuple Relations

Comments:

1) For the selection, insert, update, and delete queries, 10 variants of each query are used. For the join, aggregate, and projection queries four variants of each query are used. Queries alternate between two identical copies of each relation. For example, the selection predicates of the first four variants of Query 1 are:

```
... WHERE TENKTUP1.unique2 BETWEEN 0 AND 99
... WHERE TENKTUP2.unique2 BETWEEN 9900 AND 9999
... WHERE TENKTUP1.unique2 BETWEEN 302 AND 401
... WHERE TENKTUP2.unique2 BETWEEN 676 AND 775
```

Only one variant of each query is shown below.

2) For the indexed runs, a clustered index is constructed on the unique2 attribute and non-clustered indices are constructed on the unique1 and the hundred attribute.

3) TMP is used as a generic name for the result relation for those queries whose result tuples are stored in the database.

Query 1 (no index) & **Query 3** (clustered index) - 1% selection

```
INSERT INTO TMP
SELECT * FROM TENKTUP1
WHERE unique2 BETWEEN 0 AND 99
```

Query 2 (no index) & **Query 4** (clustered index) - 10% selection

```
INSERT INTO TMP
SELECT * FROM TENKTUP1
WHERE unique2 BETWEEN 792 AND 1791
```

Query 5 - 1% selection via a non-clustered index

```
INSERT INTO TMP
SELECT * FROM TENKTUP1
WHERE unique1 BETWEEN 0 AND 99
```

Query 6 - 10% selection via a non-clustered index

```
INSERT INTO TMP
SELECT * FROM TENKTUP1
```

```
WHERE unique1 BETWEEN 792 AND 1791
```

Query 7 - single tuple selection via clustered index to screen

```
SELECT * FROM TENKTUP1
WHERE unique2 = 2001
```

Query 8 - 1% selection via clustered index to screen

```
SELECT * FROM TENKTUP1
WHERE unique2 BETWEEN 0 AND 99
```

Query 9 (no index) and **Query 12** (clustered index) - JoinAselB

```
INSERT INTO TMP
SELECT * FROM TENKTUP1, TENKTUP2
WHERE (TENKTUP1.unique2 = TENKTUP2.unique2)
  AND (TENKTUP2.unique2 < 1000)
```

Query to make Bprime relation

```
INSERT INTO BPRIME
SELECT * FROM TENKTUP2
WHERE TENKTUP2.unique2 < 1000
```

Query 10 (no index) and **Query 13** (clustered index) - JoinABprime

```
INSERT INTO TMP
SELECT * FROM TENKTUP1, BPRIME
WHERE (TENKTUP1.unique2 = BPRIME.unique2)
```

Query 11 (no index) and **Query 14** (clustered index) - JoinCselAselB

```
INSERT INTO TMP
SELECT * FROM ONEKTUP, TENKTUP1
WHERE (ONEKTUP.unique2 = TENKTUP1.unique2)
  AND (TENKTUP1.unique2 = TENKTUP2.unique2)
  AND (TENKTUP1.unique2 < 1000)
```

Query 15 (non-clustered index) - JoinAselB

```
INSERT INTO TMP
SELECT * FROM TENKTUP1, TENKTUP2
WHERE (TENKTUP1.unique1 = TENKTUP2.unique1)
  AND (TENKTUP1.unique2 < 1000)
```

Query 16 (non-clustered index) - JoinABprime

```
INSERT INTO TMP
SELECT * FROM TENKTUP1, BPRIME
WHERE (TENKTUP1.unique1 = BPRIME.unique1)
```

Query 17 (non-clustered index) - JoinCselAselB

```
INSERT INTO TMP
SELECT * FROM ONEKTUP, TENKTUP1
WHERE (ONEKTUP.unique1 = TENKTUP1.unique1)
  AND (TENKTUP1.unique1 = TENKTUP2.unique1)
  AND (TENKTUP1.unique1 < 1000)
```

Query 18 - Projection with 1% Projection

```
INSERT INTO TMP
SELECT DISTINCT two, four, ten, twenty, onePercent, string4
FROM TENKTUP1
```

Query 19 - Projection with 100% Projection

```
INSERT INTO TMP
SELECT DISTINCT two, four, ten, twenty, onePercent,
tenPercent, twentyPercent, fiftyPercent, unique3,
evenOnePercent, oddOnePercent, stringu1, stringu2, string4
FROM TENKTUP1
```

Query 20 (no index) and **Query 23** (with clustered index)

Minimum Aggregate Function

```
INSERT INTO TMP
SELECT MIN (TENKTUP1.unique2) FROM TENKTUP1
```

Query 21 (no index) and **Query 24** (with clustered index)

Minimum Aggregate Function with 100 Partitions

```
INSERT INTO TMP
SELECT MIN (TENKTUP1.unique3) FROM TENKTUP1
GROUP BY TENKTUP1.onePercent
```

Query 22 (no index) and **Query 25** (with clustered index)

Sun Aggregate Function with 100 Partitions

```
INSERT INTO TMP
SELECT SUM (TENKTUP1.unique3) FROM TENKTUP1
GROUP BY TENKTUP1.onePercent
```

Query 26 (no indices) and **Query 29** (with indices) - Insert 1 tuple

```
INSERT INTO TENKTUP1 VALUES(10001,74,0, 2,0,10,50,688,
1950,4950,9950,1,100,
'MxxxxxxxxxxxxxxxxxxxxxxxxxGxxxxxxxxxxxxxxxxxxxxxxxC'
'GxxxxxxxxxxxxxxxxxxxxxxxxxCxxxxxxxxxxxxxxxxxxxxxxxA',
'OxxxxxxxxxxxxxxxxxxxxxxxxxOxxxxxxxxxxxxxxxxxxxxxxxO')
```

Query 27 (no index) and **Query 30** (with indices) - Delete 1 tuple

```
DELETE FROM TENKTUP1 WHERE unique1=10001
```

Query 28 (no indices) and **Query 31** (with indices) - Update key attribute

```
UPDATE TENKTUP1
SET unique2 = 10001 WHERE unique2 = 1491
```

Query 32 (with indices) - Update indexed non-key attribute

```
UPDATE TENKTUP1
SET unique1 = 10001 WHERE unique1 = 1491
```