Quickly Generating Billion-Record Synthetic Databases

Jim Gray, Prakash Sundaresan
Digital San Francisco Systems Center, 455 Market, San Francisco, CA., 94105
{Gray, Prakash}@SFbay.enet.dec.com
Susanne Englert
Tandem Computers Inc., 19333 Vallco Parkway, Cupertino, CA, 95014
Englert_Susanne @ Tandem.com.
Ken Baclawski
Computer Science, Northeastern University, 360 Huntington Av. Boston, MA. 02115
kenb@ccs.neu.edu
Peter J. Weinberger
Bell Laboratories, 600 Mountain Ave, Murry Hill, NJ., 07974
pjw @ research.att.com

Abstract: Evaluating database system performance often requires generating synthetic databases – ones having certain statistical properties but filled with dummy information. When evaluating different database designs, it is often necessary to generate several databases and evaluate each design. As database sizes grow to terabytes, generation often takes longer than evaluation. This paper presents several database generation techniques. In particular it discusses:

(1) Parallelism to get generation speedup and scaleup.

(2) Congruential generators to get dense unique uniform distributions.

(3) Special-case discrete logarithms to generate indices concurrent to the base table generation.

(4) Modification of (2) to get exponential, normal, and self-similar distributions.

The discussion is in terms of generating billion-record SQL databases using C programs running on a shared-nothing computer system consisting of a hundred processors, with a thousand discs. The ideas apply to smaller databases, but large databases present the more difficult problems.

Table of Contents

1. Introduction

Evaluating database system performance often requires generating large synthetic databases – ones with certain statistical properties, but otherwise filled with dummy information. When evaluating different database designs, it may be necessary to generate several databases and evaluate each design. As database sizes grow to terabytes, generation often takes longer than evaluation.

Large database load or generation operations last for more than a week. The goal here is to quickly generate a large database by using parallel algorithms and execution. To make the problem concrete, the goal is to generate a billion record `ACCOUNTS` table for the TPC-A benchmark [TPC]. Generating and loading this table using sequential algorithms would take several days. The goal here is to invent algorithms and techniques that generate this billion-record table and its indices in an hour.

In outline, the paper first postulates a model of parallel computer hardware and software, so that we can quantify the performance of each algorithm. Then, the paper shows how to convert a sequential load to a parallel load by partitioning the job and forking a process-per-partition. Next the tasks of synthetic data generation are investigated. Parallel algorithms are given for generating dense-unique-pseudo-random sequences, and for generating indices on these sequences. After that, the paper investigates generating non-dense non-uniform distributions with special attention paid to Zipfian and self-similar distributions.

First consider parallel computer architecture and the associated performance and cost model .

2. The Computation Model

We assume a shared-nothing computer architecture typified by the Tandem, Teradata, or Gamma machines, by workstation clusters from DEC, IBM, HP, Novell, and Sun, and by processor arrays like the Intel Hypercube [Stonebraker, Horst, Teradata, DeWitt 1, DeWitt 3]. In these systems, each processor has a private memory and one or more discs. The processors are connected via a high-speed network and processes communicate via messages. Parallelism and minimal process interaction are major design goals in these systems. Shared-disc systems like IBM's Sysplex and Digital's VMSclusters are gravitating toward this shared-nothing architecture as the number of processors grows.

The ideas presented here apply to a spectrum of execution environments – a *many-small environment* of hundreds of processors typified by Teradata and Gamma, and a *few-big environment* of tens of processors typified by Tandem and VMScluster. In Table 1, the two systems each have 3 BIPS (billion instructions per second[1]) of processing power, 10 GB of RAM, and 1 TB of disc storage (1000 discs). The prices are estimates: 100 $/MIPS, 30 $/MB RAM, and 1 $/MB disc. These approximate 1994 prices.

Table 1: Two large database machines of 1994, each costing 1.6M$.

	MANY-SMALL PROCESSORS (100)		FEW-BIG PROCESSORS (10)	
	CPUS + MEMORY	DISCS	CPUS + MEMORY	DISCS
number:	100 X (30 MIPS+100 MB)	1000 X (1 GB/disk)	10 X (300 MIPS+1 GB)	1000 X (1 GB/disk)
sum	3 BIPS +10 GB	1 TB	3 BIPS + 10 GB	1 TB
price	100 x (3 k$ + 3 k$)	1000x 1k$	10 x (30 k$ + 30 k$)	1000 x 1 k$
sum price:	600 k$	1 M$	600 k$	1 M$
total price	**1.6M$**		**1.6M$**	

The ideas here apply to both architectures – but to simplify the discussion, the many-small-processors design is assumed here.

In this presentation, CPUs are named by the numbers 0, 1,..., 99. Discs are attached to particular CPUs, and are correspondingly named D$*ddd* where *ddd* is the disc number [0...999]. For example, disc 223 is named D$223 and is the third disc of processor 22.

[1] The correct processor speed term is SPECint, but that is approximately a MIPS (million instructions per second).

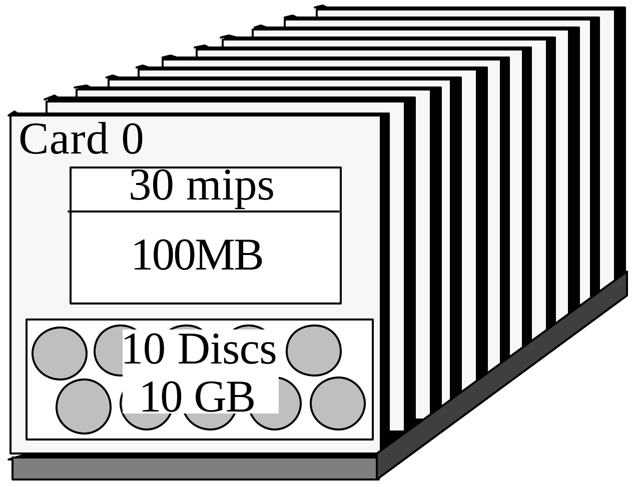


Figure 2. A diagram of a many-small computer cluster consisting of 100 "cards" each containing a moderately powerful processor equivalent to the PC or workstation of 1993. The "card" includes the processor, its communication hardware, its memory and a small array of discs. The cards are inter-connected by a high speed interconnect shown as the "base-plane". It gives 1 GB/s point-to-point connectivity between any pair of processors.

For simplicity, assume that the data of a table or index is range-partitioned among all the 1000 discs in partitions of equal size. Each system has a slightly different syatax for table partitioning. To be concrete, we use Digital's Rdb syntax [Hobbs]. Partitions are called storage areas in Rdb and are often named by the disc on which they reside. Using Rdb syntax, the ACCOUNTS table of the TPC-A Benchmark [TPC] on the many-small system would be defined by:

```
create table ACCOUNTS ( ID            unsigned integer     not null,
                        BALANCE       decimal(12,2)        not null,
                        CUSTOMER      unsigned integer     not null,
                        FILLER        character(92)        not null,
                        primary key (ID)                                        (1)
                        )
create storage map PARTITION for ACCOUNTS store using (ID)
             in  D$000 with limit of   0999999,
             in  D$001 with limit of   1999999,
             in  D$002 with limit of   2999999,
              •••
             in  D$997 with limit of 997999999,
             in  D$998 with limit of 998999999,
   otherwise in  D$999;
```

When an application selects, inserts, updates, or deletes a record of the ACCOUNTS table, the SQL system locates the record in the correct partition. The application is unaware of the partitioning. Most SQL systems provide a variant of this transparent partitioning.

The computation model is fairly simple. A sequential record read or insert costs 5,000 instructions and a fraction of an IO. Algorithms here avoid anything but sequential record reads and writes because sequential operations can run at disk device speed (5 MB/second or 50,000 records per second), while random disk operations run a thousand times slower (50 IO/sec = 50 records/second) due to seek and rotational latencies. One-way process-to-process messages have a fixed cost of 3K instructions, and a marginal cost of one instruction per byte [Uren, Kronenberg, Thekkath]. Some systems cost ten times as much for such services. In any case, the algorithms here minimize sending messages. If necessary, they send a few large messages rather than many small ones. These computational costs are summarized in Table 2.

Table 3: Cost of basic operations:

record size	100 bytes
sequential SQL record read or insert	5,000 instructions + 0.01 IO
random SQL record read or insert	25,000 instructions + 1.00 IO
disk sequential read/write rate	5 MB/sec= 50,000 records/sec
disk random read/write rate	50 IO/sec = 50 records/sec
cost of a one-way M-byte message	3 k+ M instructions (speed of light latency is minimal)

3. Sequential Database Generation

The discussion begins by showing how to sequentially generate and populate a table. This algorithm is then generalized to one that generates each partition in parallel. To make the discussion concrete, the following sections take generating the TPC-A ACCOUNTS table as a running example [TPC]. The table will have one billion hundred-byte records (.1 TB) partitioned among the 1000 discs as described in the data definition Program (1) above. Each of the one thousand discs will store 100 MB of data as a B-tree [Knuth]. Since B-trees are only 69% full at equilibrium, each disc will use 150 MB of storage to hold it's B-tree. This is well below the 1 GB capacity of small discs. The remainder of the disk space stores data from other tables.

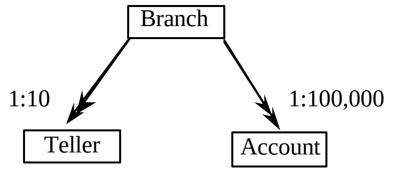
The ACCOUNTS table above can be sequentially populated by the following simple SQL + C program[2]

```
/* sequentially load records with key in [base,base+count) into tablename       */
void sequential_load(       char *tablename,     /* name of table to be loaded    */
                            long base,           /* start key of load             */
                            long count)          /* number of keys to load        */
   {
   exec sql begin declare section;
   long key;                                                                    (2)
   exec sql end declare section;
   for (key = base; key < base + count; key++)
        exec sql insert into :tablename values(:key, 0, 0, "");
   }
```

In fact the entire database for a 10,000 tps-A database [TPC] can be loaded by the following program:

```
void sequential_load("accounts", 0, 1000000000);                                (3)
void sequential_load("tellers",  0,     100000);
void sequential_load("branches", 0,      10000);
```

Program (3) shows how to sequentially generate several tables with related statistics: the idea is that the branch:teller:account cardinalities should be in the ratios 1:10:100,000, as in the schema: The entity-relations ship diagram (4) shows this relationship. Branch B has the ten tellers: 10B, 10B+1, ...; and has the hundred thousand accounts: 100000B, 100000B+1,.... The partitioning of these tables would be defined to give each of the thousand discs ten branch records, one hundred teller records, and one million account records. With 1,000 discs, disc i would get branch numbers [i•10^1..(i+1)•10^1-1], teller numbers [i•10^2..(i+1)•10^2-1] and account numbers [i•10^6..(i+1)•10^6-1].



(4)

[2] A later section will show how to fill in the customer number with a unique and uncorrelated customer ID. The customer number should be unique and uncorrolated with the account number. Here it is set to zero in all records. Also, standard SQL does not allow table names to be host variables, but this and later programs assume it is allowed. One would have to use dyanic SQL to get this effect in a standard SQL system.

4. Parallel Database Generation

The ideas in Programs (2) and (3) are fine for loading small tables (less than a million records), but they use a single processor and so run one hundred times slower than an algorithm that divides the task into a hundred smaller ones each running in parallel on a separate processor. Program (3) runs at 6,000 records/second given the performance assumptions of Table 3 (5,000 instructions per insert, 30 MIPS, implies 6,000 inserts per second.) At that rate, the billion-record load would take almost two days. The same load could run in twenty minutes if done in parallel on the hundred-processor cluster described by Figure 2 and Table 3.

Parallel algorithms require a way to create processes on specific CPUS. Assume that a process can be created in a cpu by calling[3]:

```
int fork(int  cpu)      /* creates a process in the cpu and returns it's id         */
```

The `fork` procedure is much like the UNIX `fork()` [Tanenbaum] but has a parameter specifying the `cpu` in which the process is to be created. As with UNIX, the `fork` returns zero to the child process, but returns the process identifier to the parent (forking) process. The child is a clone of the parent, executing the same program with the same current state, but a completely separate process environment.

Program (4) below shows how to convert the sequential load of Program (2) above to a parallel loader. A parallel loader proceeds by spawning a load process in each processor. The spawning time is minimized by forking a binary tree of processes, forking 2^n processes at depth n of the tree. Each node follows the logic:

(1) If I am not a leaf, fork my left child and right child.

(2) Load my partition.

Figure 4 illustrates the idea. The logic is designed for a cluster of 2^N nodes, but it works on smaller clusters of M nodes. If each fork takes about one second (a high estimate), the entire startup of one hundred processes will complete within eight seconds. Assuming the forking logic copies code and data from the forker, there will be no bottlenecks at startup.

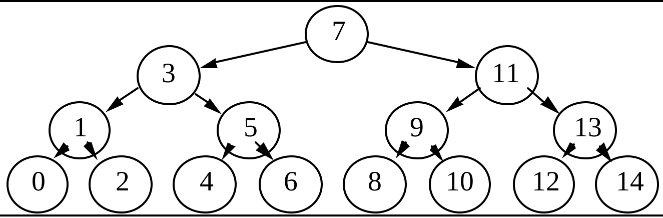


Figure 4. The process forking logic for a cluster of 15 processors. cpu 7 forks loaders into cpus 3 and 11, then cpu 7 proceeds to load partition 7.

[3] What is really needed is more akin to Unix `exec` than to `fork,` and has many more parameters.

The pseudocode for parallel load is:

```
#define  CPUS 100                                                                 (5)
/* parallel load records into tablename                                    */
/* first fork right and left child processes if right != left              */
/* then sequentially load the partitions of this cpu                       */
void parallel_load(char *   tablename,    /* name of table to be loaded          */
                            long records, /* number of records to load           */
                            long left,    /* cpu of left fork subtree            */
                            long right,   /* cpu of right fork subtree           */
                            integer depth);/* recursion depth in forking         */
   { long my_cpu    = floor((left+right)/2);
   long part_size = floor(records/CPUS); /*
   long him;                             /* id of forked process                 */
   depth++;                              /* increase recursion depth             */
   if ( depth == 1 )                            /* return after forking root process   */
        { if  ( fork(my_cpu))  return; }; /* in center cpu.                      */
   /* spawn left subtree of processes                */
   if ( left < my_cpu )
        { him = fork( floor( (left + my_cpu-1) / 2) );
          if (him == 0)    /* child code                                         */
               {parallel_load(tablename, records,left,my_cpu-1,depth); return; }
        }
   /* spawn right subtree of processes               */
   if ( my_cpu < right )
        { him = fork( floor( (my_cpu + 1 + right) / 2) );
          if (him == 0)    /* child code                                         */
               { parallel_load(tablename, records,my_cpu+1,right,depth); return; }
        }
   /* fill the partition of this process                                        */
   sequential_load(tablename, my_cpu * part_size, part_size);
   };                                    /* end of parallel_load()               */
/* invoke parallel loader for a billion account records on 100 cpus             */
void = parallel_load("accounts", 1000000000, 0, CPUS - 1, 0);
```

This code is easily modified to fork a process-per-disc rather than a process-per-processor if the discs are the bottleneck. Notice that each generator process uses the same table name to generate the data (transparency). The table partitioning criterion causes the records all to go to each loading process's local disc.

Parallelism often suffers from problems of startup, interference, and skew [Gerber, Smith]. Program (5) minimizes startup problems by parallel forking. Once the load process begins, the underlying system acquires locks on partitions rather than on whole tables – at least that is the way many SQL systems work. So, each partition loader can proceed in parallel and in isolation. The load operation is typically not covered by transaction protection, so the recovery log is not a bottleneck-- rather it uses the old-master-new-master recovery technique of dumping a copy of the table when the load completes. Given these arguments, startup, interference, and skew should not be a problem for the parallel load.

Using the assumptions of Figure 2 and Table 3, the algorithm should generate 600,00 records per second (60 MB/s) and generate1B records in less than 30 minutes.

5. Dense Unique Random Data Generation

The parallel data generator of the previous section correctly generated the sequential account numbers but it did not generate the customer numbers – it just left them as zeros. Customer number is an example of a general problem. Generating synthetic databases often requires a sequence of numbers (i.e. field values) with the following properties:

Dense: All the integers in [0..N] appear in the sequence.

Unique: Each integer appears exactly once.

Random: The sequence appears to be "random" (is pseudo-random).

These properties are often needed to make the cardinalities of selection expressions and join expressions predictable – for example each customer should have exactly one account. Some applications need synthetic data that is not uniformly distributed. Section 8 gives some ways to transform uniform distributions into Gaussian, exponential, Zipfian, and other distributions.

We know of several ways to generate dense-unique-random numbers in the range *[1..N]*. The original generator for the Wisconsin Benchmark [Bitton 1] kept an initially zero *bitmap* of length N and used the system random number generator to pick the next free element for the series. This algorithm was replaced in the original ASAP generator [Bitton 3] by a shuffle that built an array of pairs *< (i, random()) | i=1,... , N >*. The array was then sorted on the second element to produce a shuffle of the first element.

The bit filter algorithm uses order *N* space and order N^2 time (about N tries are required to set the last bit in the bitmap). The shuffle algorithm takes *NlogN* time and linear space, so is clearly superior to the bit filter. If space is not an issue, shuffle is a good way to generate a dense-unique random series. But for large databases or if several independent series are to be generated a more space-efficient algorithm may be needed. The obvious choice is to generalize the shuffle to a sort:

Sort: Create a SQL table of two columns containing the dense sequence 1..*N* and a random sequence based on a popular random number generator. Then the dense sequence is ordered by the random sequence. Program (7) demonstrates this.

```
   exec sql create table T (sequence integer, rand integer); /* create temp table      */
   for (i =  1;  i <= N; i++)                         /* fill it with pairs     */
        exec sql insert into T values (:i, :random());/*                        */
        /* note that sequence values are dense and unique.                      */
   /* If ordered by the rand field,  they will be random.                       */

   exec sql declare cursor answer for          /* define a sql cursor to       */
        select sequence from T order by rand;  /* get data in the random order */

   exec sql open answer;                       /* read the table via that cursor      */
   for (i = 0; i < N; i++)                     /* set next_value from          */
        { exec sql fetch answer into :next_value;/* next record                */
        /* process next value                                                   */
        }                                      /*                              */
   exec sql close answer;                      /*                              */
```

This algorithm takes $NlogN$ time and $\sim\sqrt{N}$ main memory space (sorting) and $\sim N$ disc space (storage of table). For a billion records, the many-little configuration can do the sort in about thirty minutes and the job in less than an hour.

Such schemes are somewhat inconvenient because they construct a set of files to drive data generation. It would be more convenient to have a simple subroutine that could generate the next element of the desired sequence in constant time and space (say 25 instructions and 100 bytes of storage). The idea for such an algorithm is to generate the numbers using a generator of the cyclic group of integers under multiplication. In essence, a random number generator is constructed for elements in the desired range. The algorithm is:

Pick a prime P larger than N and a generator G for the multiplicative group modulo P. Then the series is:

$$<G^i \bmod P \mid i = 1,...,P \quad and \quad (G^i \bmod P) = N>$$

and the program to generate the series is:

```
#define  P xxx;                              /* see Table 5 for good values   */    (8)
#define  G xxx;                              /* see appendix 1 for good values      */
static   seed = G;                           /* start the seed at G           */
long next_value(long N)                      /* function to compute next value */
   { seed = (G * seed) % P ;                 /* seed = next in series mod prime      */
   while ( seed > N ) seed = (G * seed) % P;                                 /*
discard all > N     */
   return seed;                              /* return new value              */
   }                                         /* end of generator              */
```

This scheme, due to Gray and Englert, was successfully used to generate very large "Wisconsin Benchmark" databases on the Intel Hypercube and is now the standard way to generate Wisconsin databases [DeWitt 2]. To understand how it works, consider the numbers between 1 and 10. The powers of 8 mod 11 form a dense unique sequence of these numbers: 8, 9, 6, 4, 10, 3, 2, 5, 7, 1. In general, if *N* is a prime the multiplicative group consisting of *[1..N-1]* has many generators: elements whose powers enumerate the group without repetition until they generate 1.

Clearly, the generator scheme is preferable – it takes constant time and constant space. But not just any generator will give a good pseudo-random sequence. There are many tests for "randomness". We used the spectral test recommend by Knuth [Knuth]. In his terminology, all the random number generators in this paper pass the spectral test "with flying colors" in dimensions 2 through 6. We applied the test to primes just larger than powers of ten and recommend the generators of Table 5. Section 7 describes how to use powers of 2 instead of primes.

Table 5. A list of recommended primes and generators for each decade from 10 to one billion.

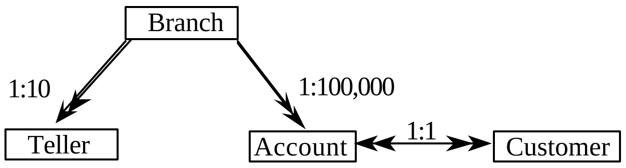
Decade	Prime	Generator
10	11	2
100	101	7
1,000	1,009	26
10,000	10,007	59
100,000	100,003	242
1,000,000	1,000,003	568
10,000,000	10,000,019	1792
100,000,000	100,000,007	5649
1,000,000,000	2,147,483,647	16807

When the prime and generator are large compared to the machine's arithmetic, one needs to use a technique shown in Program (9) due to Schrage [Schrage] to keep the results from overflowing. P and G must be chosen so that B is less than A. This will always be true if $G < \sqrt{P}$. Machines with 64-bit registers and arithmatic make this technique unnecessary.

```
#define  P xxx;                           /* see Table 3 for good values      */    (9)
#define  G xxx;                           /* of prime and generator           */
#define  A (P / G);                       /* A = prime / generator            */
#define  B (P % G);                       /* B = prime mod generator          */
static   seed = G;                        /* start the seed at G              */
long next_value(long N)                   /* function to compute next value   */
   { long    seed_over_A = seed / A;      /* compute the components of seed   */
   long      seed_mod_A  = seed % A;      /* compared to A = (P/G)            */
   do                                     /* loop if next is bigger than N    */
     {                                    /* Use Schrage's function to        */
     seed = (G * seed_mod_A ) - (B * seed_over_A) ; /* compute G*seed mode P  */
     if ( seed < 0) seed = seed + P;      /*             without overflow     */
     } while ( seed >= N )                /* discard all >= N                 */
   return seed;                           /* return new value                 */
   }                                      /* end of next_value()              */
```

6. Generating Random Data

The ideas of the previous section can now be applied to generate a complete table. Program (3) above gave an example of generating several tables with related statistics: the idea there was that the branch:teller:account cardinalities should be in the ratios 1:10:100,000, as in the schema (4). This is a general phenomenon, but the requirements are often more complex. One requirement that was skipped in Program (2) was that the customer id field be unique and be uncorrelated with the account id – rather it was just filled with zeros. The requirement is that each customer have a unique bank account as shown in (9).



(9)

Using the ideas of the pervious section, Program (2) can be refined to generate each partition of the account table in parallel, including the random-unique-dense customer number as follows:

```
/* sequentially load records into i'th partition of tablename                    */
void sequential_load( char *tablename,          /* name of table to be loaded    */
                      long records,             /* number of records in table    */
                      long count,               /* # records in each partition   */
                      long part_no)             /* number of partition to load   */
   {                                            /*                               */
   long i, j;                            /* loop control variables               */
   long base = part_no * count;                 /* start key for this partition  */
   long parts = records / count;                /* number of partitions in file  */
   exec sql begin declare section;                     /* SQL variables          */
   long key, customer;                          /* account id, customer id       */   (10)
   exec sql end declare section;                /*                               */
   for (i = 0; i < part_no; i++)                /* the i'th processor skips to   */
        customer = next_value(records);         /* p(i) in generator's sequence         */
   for (key = base; key < base + count; key++)  /* now generate the partition    */
        {customer = next_value(records) - 1;    /* get next customer number      */
        exec sql insert into :tablename  ( id, balance, customer, filler) /*     */
                           values(:key, 0,:customer,  "");/*generate 1/n'th      */
        for (j = 0; j < CPUS; j++)              /* skip seed to p(i+CPU)         */
             customer = next_value(records);    /* i.e.  skip p(j) of others     */
   }    }                                       /* end of sequential_load()      */
```

The generator of the i'th cpu is invoked as:

```
        void sequential_load("accounts", records, records/CPUS, i);
```

Each of the one hundred processors will compute the same random series based on the `next_value(records)` procedure. But each generator will only use one hundredth of the series. If the series is s_0, s_1, s_2,....,$s_{999999999}$, the i'th cpu will use the subsequence s_i, $s_{1cpus+i}$, $s_{2cpus+i}$, $s_{3cpus+i}$, $s_{4cpus+i}$,... Each of these subsequences is random and unique, and the union of them is dense. The i'th generator begins by calling `next_value()` "i-1" times to skip over the values of the other generators. Then after inserting a record, the generator calls the `next_value()` `CPUS` times, but only uses the last value returned. There is a lot of wasted work in this design: the series is computed one hundred times and each generator only uses 1% of the values it generates. The premise is that calls to `next_value()` are cheap (~25 instructions) so that 100 calls (2500 instructions) is small compared to the insert cost (~5000 instructions). The generator should produce 1B records in less than 1 hour on the many-small configuration.

Scaling this algorithm to thousands of generators requires a variation that has less wasted work. One might partition the series into 1000 segments and precompute the starting point p_i for each partition. These values could be stored in a global array. Then each partition generator would start at s_x, for some x, and would use the sequence s_x, s_{x+1}, s_{x+2},... and no calls to `next_value()` would be wasted.

A different approach avoids this pre-computation and minimizes wasted computation. Table 5 shows that for large N, P can be chosen just slightly larger than N (within 1% of it). Suppose we generate a database of $P\text{-}1$ elements rather than N elements. In that case, no members of the series $<G^i \bmod P \mid i = 0,...,P>$ would be discarded. In turn this means that each partition can compute its next element by multiplying the previous element by $G^{CPUS} \bmod P$. Let $n = ?\, N/CPUS\, ?$and assign the following series to the j'th partition for j = 0,...,$CPUS$ - 1:

$$<G^{CPUS \cdot i+j} \bmod P \mid i = 0,...,n\text{-}1>$$

This series is very easy for partition j to compute. First it computes the first element $A\ ?\ G^j \bmod P$ and then $B\ ?\ G^{CPUS} \bmod P$. These two numbers can be computed in $ln(CPUS)$ multiplies and divides. Then the j'th partition uses the series:

$$<(A \bullet B^i) \bmod P \mid i = 0,...,n\text{-}1>.$$

In this approach, there is no need to precompute a partition table and there are no wasted calls to `next_value()`. If we accept this relaxed definition of partitions (the last partition may be slightly larger than the others and some elements may be a little larger than N), then it will turn out that computing indices is much easier

These techniques can easily generate tables with 1:N relationships. Suppose, as in (9), the BRANCHES table is to have 1,000 records and the TELLERS table is to have 10,000 records. If, P is chosen as 1,009 and G is chosen as 229 (as Table 5 recommends), then BRANCHES can be generated as in Program (10). By using the same prime and generator for the BRANCH field of the 10,000 record TELLERS table, the TELLERS table will have exactly 10 tellers for each branch identifier. More generally, each record in one table will match the value of M records in the second table if the second table has M times as many records, and the "join-fields" of the two records use the small-table generator.

7. Generating Indices on Random Data

All the programs so far have carefully (but implicitly) generated data in primary key order, the next generated record is placed right after the previous record in the B-tree or other clustering mechanism. This means that the programs have generated the data in sequential order and so that disc IO time has not been an issue. Modern discs can absorb data at 5 MB/s – with the possibility of much higher data rates if striping is used [Kim]. Assuminmg 100 byte records, this is 50,000 records per second. If each generated record went to a random disk page, data rates would drop by a factor of 1000 to 50 records per second since each record would cause a seek, rotate, a read transfer and then a rotate and a write transfer. On 1993 discs, each random disk IO consumes about 20ms of disc time and the rate is at most 50 IO/s. So, it is *essential* that records be generated in sequential order unless the entire table can fit in main memory.

Program (10) generates the `ACCOUNT` table in `ACCOUNT.ID` order; but it generates the `ACCOUNT.CUSTOMER` field in random order. Applications often need an index on such random fields. For example, an `ACCOUNT.CUSTOMER` index would allow quick lookup of a customer's `ACCOUNT` given the customer number. Such an index is defined in SQL by[4]:

```
     create unique index account_customer on account (customer);
```

The index is actually a table with the schema:

```
   create table account_customer (      customer      unsigned integer not null,
                                  id            unsigned integer not null,          (12)
                                  primary key (customer),
                                  foreign key (id) references account(id)
                                  );
```

If *(i,j)* is a record in the `account_customer` index then;

i is the j'th value returned by `next_value(records)`

or using *G* and *P* as defined in (8):

$i ? G^j \mod P$.

More formally, *j is the discrete logarithm of i* [Coppersmith].

How can this index be generated in parallel with linear speedup and scaleup? One could compute the discrete logarithm, but [Coppersmith] indicates that each computation would be millions of instructions. Three schemes can be used:

[4] As usual, we assume this table (index) is uniformly partitioned among the 1000 discs. The definitions of the partitions are omitted here for simplicity.

(1) **Scan-and-sort**: Read the base table in parallel, projecting out the two desired fields, and parallel sort the result into the target index. In SQL this would be expressed as:

```
insert into account_customer
     select customer, id
     from account
     order by customer;
```

(2) **Generate-and-sort:** In each processor, generate the index data to be stored by that processor's disks, sort it, and then insert it into the local index partitions.

(3) **Compute:** Compute the discrete logarithm quickly and generate the index in the same way one generates the base table.

The index is one billion records of eight bytes each, 8 GB in all. So, each of the 100 processors must deal with an 80 MB partition of the index. This will just fit in each processor's 100 MB memory. The scan-and-sort approach (scheme 1 above), the processors can generate the index data in parallel to the data generation, sending index records to the appropriate partitions (cpus) as the base table is generated locally. The receiving processors can sort the indices locally in their memory as the data arrives or is generated. This is a credible and scaleable technique, needing about 10 MB/s network bandwidth to move the 8 GB from source to destination for a one-hour job. But the technique is memory intensive, just barely fitting in the processor memories. If the table keys were larger of if there were more indices, then the scan-and-sort technique would require a disk-based sort.

Scheme 2 is a more cpu-intensive sorting scheme - but uses no network messages. Each processor generates the entire base-table sequence, and extracts the index subsequence that applies to the local processor. In particular, if each partition has R index records and if the whole sequence is $s_1, s_2\ s_3 \ldots$ then the *i'th* processor uses the subsequence:

$$<<s_j, j> \mid iR = s_j\ ?\ ?(i+1)R)>$$

These are the index entries for the i'th partition. They are then sorted on the s_j attribute and inserted into the local index partition in sequential order. The following program shows this cpu-intensive enumerate-and-sort algorithm.

```
/* global variables: seed , generator, CPUS, my_cpu                          */
/*  R is records per partition                                               */
void index_load(long records)
   { long R = records / CPUS;                 /* R records per partition      */
   long my_first_record = my_cpu * R;    /* base of index partition           */
   long i,j = 0, customer;               /* working variables                 */
   exec sql begin declare section;            /*                                     */
   struct { long account;                /* array holding in-memory index to be */
            long customer;               /* sorted on the "random" customer id  */
          } sorted [R];                  /*                                   */  (13)
   exec sql end declare section;         /*                                   */
   /* fill in the array with the unsorted values                                    */
   for (i = 0 ;  i < records; i++)       /* for each account number           */
        { customer = next_value(records);      /* get next customer number     */
        if ( (my_first_record = customer ) &  /* if customer # is in the range  */
             (customer < my_first_record + R) ) /* of this partition, use the entry*/
             {sorted.account[j] = i; sorted.customer[j] = customer; j++} /*     */
        };                               /* assert: now sorted[i] = G inverse  */
   /* sort the array on the second attribute (customer)                              */
   sort(sorted) on sorted.customer;      /* sort the array on customer attribute*/
   for (j = 0; j < R; j++)                    /* copy the array to the index   */
   exec sql insert into account_index values (:sorted.customer[j] , :sorted.account[j]);
   };
```

The generation step should take 30 instructions per iteration, and there are 1 billion iterations, so it will take 1000 seconds on a 30 MIPS computer. The sort deals with ten million records. At 30 instructions per compare/exchange, an *nlog(n)* sort will need about 300 seconds. Once that is complete, the write of the data in bulk to the index (at 1000 instructions/insert) should take 300 seconds. This adds up to about thirty minutes. In summary, indices for large tables can be built in parallel while the base tables are being built. The generator described here can run in parallel with the base table generation if sufficient processors and memory are available.

The third index technique involves quickly computing discrete logarithms. That is, given alternate-key value k, quickly compute primary key value i such that

$$k \; ? \; G^i \bmod P. \tag{14}$$

Solving this problem for arbitrary k when P is a large prime is believed to be quite difficult. Indeed, this is what makes some cryptographic protocols seem secure. Even for smaller primes, around a billion, each discrete logarithm calculation takes about $\sqrt{p}$ time and space. The sorting algorithm above would be faster.

Picking P as a power of 2, say 2^n, allows computing discrete logs in n steps (*logP* time) and constant space. The following equation is the key to finding the discrete log of k when P is a power of 2:

$$G^{2^i} \; ? \; 1 + 2^{i+1} (\bmod 2^{i+3}). \tag{15}$$

The problem is that the values of the series $G, G^4, G^{16},.. , G^{2^i},...$ are all congruent to *1 mod 4* (all their binary representations end in "01"). The solution is to divide the numbers by 4 to get a dense series. The following code computes the discrete log of *k* (the 2-adic log):

```
#define   POWER       32            /* the P will be 2^32              */
#define   G           37117         /* Generator for P from Table 6    */
static long P      = pow(2,POWER);  /* compute P                       */
static long P_MASK = P - 1;         /* mask to avoid mod P division    */

/* do mod P multiplication, in the *4 + 1 space.  The equation is:     */
/*     ( a * 4 + 1) * ( b * 4 + 1 ) = ( a + b + 4 * a * b ) * 4 + 1   */  (16)
long mul(long a, long b)            /*                                 */
   return ((a + b + 4 * a *b) & P_MASK);  /*                           */

/* discrete_log(k) returns the discrete log of k with respect to G     */
/*   At entry                       G^(x) = 1 + 4 * j
/*   the invariant of the loop is:  G^(x+ans) = 1 + 4 * up             */
/*                            where the last i bits of up are zero     */
long  discrete_log(long j)          /*                                 */
   {                                /*                                 */
   long up = j,                     /* up is j with G^i removed        */
   long i;                          /* index on radix bits of k        */
   long ans = 0;                    /* the target                      */
   long radix = 1;                  /* radix = 2^i in the loop below   */
   long Gpow = G;                   /* Gpow = G^2i                     */
   for ( i = 1;  i < POWER; i ++)   /* for each bit POWER              */
        { if (up  & radix)          /* if 2^i divides k                */
             { ans =  ans + radix;  /* equation (15) says add 2^i      */
               up  = mul( up, Gpow ); }/* preserve loop invariant      */
        radix = radix * 2;          /* advance radix = 2^i             */
        Gpow = mul( Gpow, Gpow );   /* advance Gpow = G^2i             */
        }                           /* end of loop                     */
   /* now up = 0 so G^(x+ans) = 1 due to the invariant.                */
   /* by Fermat's theorum, x+ans = POWER so discrete log is POWER-ans  */
   return (POWER - ans);            /*  discrete log of k mod 2^POWER  */
   }                                /*    end of discrete_log()        */
```

Table 6 is a catalog of values for POWER and G that have passed the spectral test.

Table 6. Powers of 2 and generators to compute discrete logs.

Field	**(max value)**	**POWER**	G	**G^{-1}**
$1...2^{8}-1$	255	10	29	565
$1...2^{9}-1$	511	11	29	565
$1...2^{10}-1$	1,023	12	53	541
$1...2^{11}-1$	2,047	13	53	4637
$1...2^{12}-1$	4,095	14	117	4061
$1...2^{13}-1$	8,191	15	125	30,933
$1...2^{14}-1$	16,383	16	229	48,365
$1...2^{15}-1$	32,767	17	221	55,157
$1...2^{16}-1$	65,535	18	469	77,693
$1...2^{17}-1$	131,071	19	517	31,437
$1...2^{18}-1$	262,143	20	589	329,349
$1...2^{19}-1$	524,287	21	861	747,765
$1...2^{20}-1$	1,048,575	22	1,189	2,638,637
$1...2^{21}-1$	2,097,151	23	1,653	5,577,181
$1...2^{22}-1$	4,194,303	24	2,333	12,124,469
$1...2^{23}-1$	8,388,607	25	3,381	32,611,613
$1...2^{24}-1$	16,777,215	26	4,629	51,785,021
$1...2^{25}-1$	33,554,431	27	6,565	32,056,877
$1...2^{26}-1$	67,108,863	28	9,293	260,289,669
$1...2^{27}-1$	134,217,727	29	13,093	94,679,213
$1...2^{28}-1$	268,435,455	30	18,509	561,787,013
$1...2^{29}-1$	536,870,911	31	26,253	31,247,429
$1...2^{30}-1$	1,073,741,823	32	37,117	3,730,050,133
$1...2^{31}-1$	2,147,483,647	33	52,317	766,551,637
$1...2^{32}-1$	4,294,967,295	34	74,101	6,731,589,341
$1...2^{33}-1$	8,589,934,591	35	104,581	27,095,900,237
$1...2^{34}-1$	17,179,869,183	36	147,973	5,486,951,117
$1...2^{35}-1$	34,359,738,367	37	209,173	46,427,772,477
$1...2^{36}-1$	68,719,476,735	38	296,029	114,535,788,533
$1...2^{37}-1$	137,438,953,471	39	418,341	273,639,336,365
$1...2^{38}-1$	274,877,906,943	40	591,733	411,014,549,725

1...2^{39}-1	549,755,813,887	41	836,661	1,051,514,334,749
1...2^{40}-1	1,099,511,627,775	42	1,183,221	1,540,143,871,581
1...2^{41}-1	2,199,023,255,551	43	1,673,485	2,064,054,447,557
1...2^{42}-1	4,398,046,511,103	44	2,366,509	6,563,150,205,861
1...2^{43}-1	8,796,093,022,207	45	3,346,853	34,112,825,814,573
1...2^{44}-1	17,592,186,044,415	46	4,732,789	25,355,912,192,221
1...2^{45}-1	35,184,372,088,831	47	6,693,237	80,269,716,416,221
1...2^{46}-1	70,368,744,177,663	48	9,465,541	241,407,036,416,013
1...2^{47}-1	140,737,488,355,327	49	13,386,341	3,774,646,034,285
1...2^{48}-1	281,474,976,710,655	50	18,931,141	787,472,057,538,829
1...2^{49}-1	562,949,953,421,311	51	26,772,693	105,651,650,1967,997
1...2^{50}-1	1,125,899,906,842,620	52	37,862,197	183,937,449,308,9565
1...2^{51}-1	2,251,799,813,685,250	53	53,545,221	1,969,087,082,879,949
1...2^{52}-1	4,503,599,627,370,490	54	75,724,373	11,327,587,540,653,821
1...2^{53}-1	9,007,199,254,740,990	55	107,090,317	6,288,511,001,205,061

The use of numbers near a billion strains the word size of "old" 32-bit computers. In particular, if P is bigger than 2^{16} or so, multiplication modulo P cannot be done without some programming trick. Schrage's technique, as shown in program (9), can be used to fit such arithmetic into small words [Schrage].

In summary, indices on synthetically generated data can be built in one of three ways. Scan-and-sort, generate-and-sort, or compute. The computational method has some restrictions on the size of the table and on the generator, but is the most efficient approach for large tables. The computational approach is nicely suited to parallel algorithms.

8. Generating Data Having Non Uniform Distributions

Having explained how to generate unique data, now we consider generating other data distributions. The examples above all required dense-unique values. Often, the database needs values obeying some common distribution. The size of cities, lengths of words, and frequency of words is known to follow a Zipfian distribution. Measurement errors often obey a Gaussian distribution, and the inter-arrival intervals of events often follow a Poisson or negative exponential distribution. Such domains are easily generated by skewing a uniform distribution. This section catalogs the standard distributions, and adds a little to the generation of self-similar and Zipfian distributions.

Program (10) demonstrated the simplest case, repeating some value a constant number of times in another field. It generates ten accounts per branch – repeating each branch number ten times in the `ACCOUNT.BRANCH` domain.. Suppose we wanted a the values of some field or the number of child records to follow some more complex distribution. Then the following code might be appropriate

```
create table parent ( master      integer not null,
                     rest         char(96),
                     primary key (master)
                     );
create table child ( master       integer not null,
                     detail       integer not null,
                     rest   char(92),
                     primary key (master,detail),
                     foreign key (master) references parent
                     );
/* sequentially load records with key in [base,base+count) into tablename       */
void sequential_load(       char *parent,       /* name of parent table         */
                            char *child,        /* name of child table          */
                            long base,          /* start key of load            */
                            long limit)         /* first key after load         */
   {                                            /*                              */
   exec sql begin declare section;                     /*                              */
   long master, detail;                         /* master and detail key values */   (25)
   exec sql end declare section;                /*                              */
   for (master = base; master < base + count; master++)  /* for each master key */
        {exec sql insert into :parent values(:master, ""); /* add master rec    */
        count = distribution();                 /* create that many child recs  */
        for ( detail = 0; detail < count; detail++ )  /*                        */
             exec sql insert into :child values(:master, :detail, ""); /*       */
        }                                       /* end master loop              */
   }                                            /* end sequential_load()        */
```

This code encapsulates the distribution of child cardinalities. It only remains to describe ways of generating the popular distributions. The following table presents the code to generate the distribution on the left and a graph of the distribution on the right. See [Ripley], [Jain], or [Press] for more details.

The programs below assume `randf()` returns values distributed uniformly in [0..1] and `random()` returns values distributed uniformly in [0..?] or some approximation thereof (e.g. [0..2^{32}]).

The uniform distribution

equation: f(x) = 1 x ? [0,1]

mean: ? = 0.5

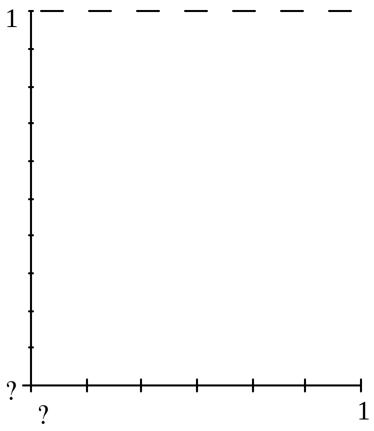
standard deviation: ? = .5

inverse distribution:

```
    x= randf() where randf() ? [0,1]
```

code:

```
double uniform()
   {return randf();}
```



Negative exponential

equation: $f(x) = \frac{1}{?} e^{-?x}$: x ? [0,8]

mean: $\frac{1}{?}$

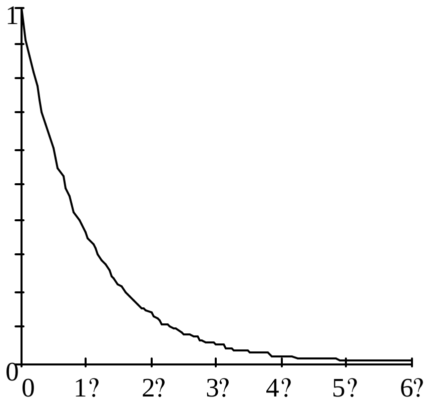
standard deviation: $\frac{1}{?}$

inverse distribution: ? ?**Error!**,?)

```
          where  random() ? [0,8]
```

code:

```
   double neg_exp(double lambda)
      { return ln(random()) / lambda; }
```



Gauss = Normal Distribution:

equation: $?(x) = \frac{1}{?\sqrt{2?}} e^{-\frac{1}{2}\left(\frac{x-?}{?}\right)^{2}}$

: x ? [0,8]

mean: ?

standard deviation: ?

inverse distribution:

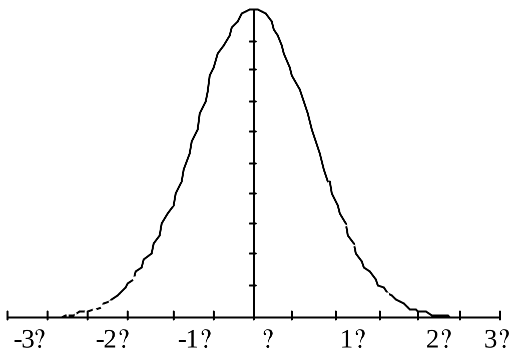
$$? \quad ? \quad ? \; ? \; ??+\frac{?}{?}\overset{12}{\underset{1}{?}} (\text{random()-0.5})$$

where randf() ? [0,1]

```
 deviation is ~.5% see [Ripley, pp 54]
```

code:

```
double gauss(double mu, sigma)
  { int i; double ans = 0.0;
  for (i = 0; i<12; i++)
    {ans = ans + (randf()) -0.5;}
  return (mu + ans/6);
  }
```



Poisson

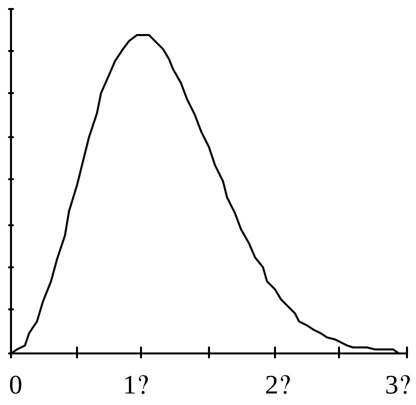
equation: $f(x) = e^{-?} \; ?\frac{?^{k}}{k!}$: k = 0, 1, ...

mean: ?

inverse distribution: [Ripley, pp 55].

code:

```
    long poisson(long lambda)
           { long  n = 0;
           double c =  pow(e,-lambda);
           double p = 1.0;
           while (p >= c)
             {p = p * randf();
              n++;}
            return (n-1);
           };
```



Self-Similar (80-20 rule)

Integers between 1...N.

The first h•N integers get 1-h of the distribution.

For example: if N = 25 and h= .10, then

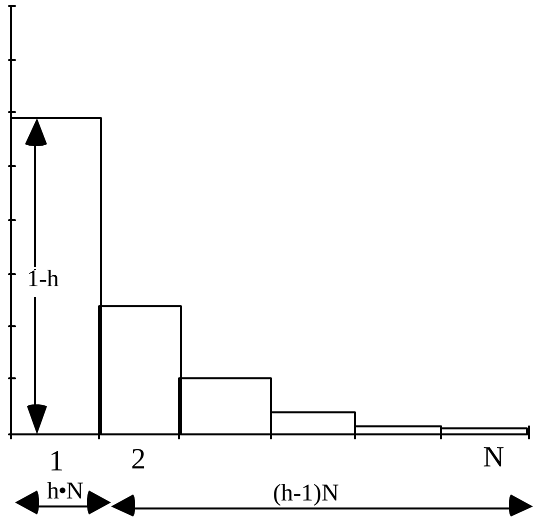
80% of the weight goes to the first 5 integers.

and 64% of the weight goes to the first integer.

code:

```
    long selfsimilar(long n, double h)
           { return (1 + (int)
      (N * pow((randf(),log(h)/log(1.0-h)
                );

        };
```



Zipf's "Law"

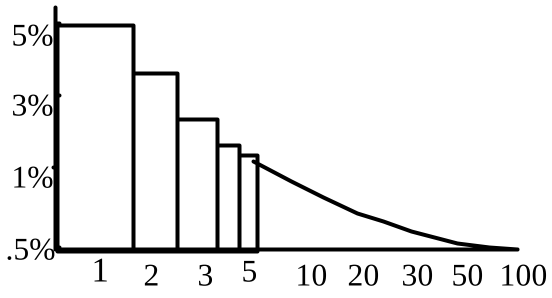
Integers between 1...N.

Integer *k* gets weight proportional to $(\frac{1}{k}.)^{theta}$

where $0 < theta < 1$ is the skew

code:

```
long zipf(long n, double theta)
  {
  double alpha = 1 / (1 - theta);
  double zetan = zeta(n, theta);
  double eta =
          (1 - pow(2.0 / n, 1 - theta)) /
          (1 - zeta(theta, 2) / zetan);
  double u = randf();
  double uz = u * zetan;
  if (uz < 1) return 1;
  if (uz < 1 + pow(0.5, theta)) return 2
  return 1 +
   (int)(n * pow(eta*u - eta + 1, alpha)
   };
```



The only "new" distribution here is the self-similar one, often used for situations following the 80/20 rule or some other highly skewed self-similar distribution. Self similar distributions have the property that within any region of the distribution, the skew is the same as in any other region. So, for example, all subranges of the 80/20 (h=.20) self similar distribution follow the 80/20 rule.

A set of values of the form 1..k is called a "hot spot" because any of the values in this set has more weight (and hence is "hotter") than any value outside the set. Self-similar distributions are characterized by the property that hot spots have a distribution similar the entire range of values. The Zipf distributions are characterized by the property that the frequency distribution is a straight line when plotted on a log-log graph.

If the hot spot is supposed to be randomly spread throughout a range of values, then it is necessary to permute the values randomly. Generating random permutations is discussed elsewhere in this paper. In principle, generating a random permutation and generating a distribution are independent problems. However, hot spot distributions like the self-similar and Zipf distributions can be permuted more easily than general distributions. The trick is to assume that the "cold" values are uniformly distributed. This greatly simplifies the generation of the permutation since it now just involves choosing the relatively small number of "hot" values. This is especially important when it is necessary to perform the computation in parallel. The "hot" values can be chosen at one node and broadcast as a table to the rest of the nodes. The algorithms above are used to compute an index into the table of "hot" values. If the index is too large, then a value is chosen uniformly at random from all possible values (ignoring whether it is already in the table of hot values). The index computations can be done independently at each node so long as the seeds for the pseudo-random number generator are chosen independently.

The number of hot values that require special treatment depends on how accurately one needs to represent the distribution in question. (One should bear in mind that the self-similar and Zipf distributions are themselves only approximations to what is observed in actual systems.) As an example, consider a self-similar distribution following the 95/5 rule. A table of just 313 out of a billion possible values would account for over 77% of the weight of this distribution.

The program presented here to generate a Zipf distribution uses constants *alpha*, *zetan* and *eta* derived from *theta* and n. The function *zeta* returns the sum

$$(1/1)^{theta} + (1/2)^{theta} + \ldots + (1/n)^{theta}.$$

The approximation uses the same technique as that in Knuth (volume 3, page 398), but corrects the weight assigned to the first two values to get a more accurate approximation.

It is commonly thought that self-similarity and the Zipf distribution are the same, or at least close. This misconception apparently stems from a misleading approximation made in Knuth (volume 3, page 398). Knuth's approximation is adequate for the statistic being computed there but should not be construed as asserting that the self-similar and Zipf distributions are close in the usual probabilistic sense. In particular, other statistics can yield very different results for the self-similar and Zipf distributions. However, Knuth's calculation can be modified to produce a reasonable approximation as we noted above.

The log-log graph of Zipf's distribution with parameter 0.5 is shown above. It shows the largest weight on 1, the second largest on 2, and so on.

9. Summary

This paper first showed how to convert a simple sequential load into a parallel load – turning a two-day task into a one-hour task. It then explored the ways to generate synthetic data. At first it focused on generating the primary keys of records and values uncorrelated to these keys: dense-unique-pseudo-random sequences. Then, attention turned to building indices on these synthetic tables – either by sorting, or by using discrete logarithms. By careful selection of generators, the discrete log problem is tractable and indices can be quickly generated within the 1-hour limit we set for the billion-record load.

The paper then looked at skewed distributions. It presented the standard ways to generate uniform, exponential, normal, and Poisson distributions. It went into more detail on the new topic of self-similar and Zipfian distributions.

Using these techniques, one can generate billion-record databases in an hour, and a two terabyte databases per day.

10. Acknowledgments

This paper has been in-progress since 1987. During that time, many people have contributed ideas to it. Some of the inspiration for the work came from Dina Bitton and Jeff Millman who showed us the data generators they used in their DBstar product. Dave DeWitt was an early user of these algorithms and has been a source of ideas and encouragement. Betty Salzberg explained the concept of discrete logs to Jim Gray an put him in contact with that community.

11. References

[Bitton 1] Bitton, D., DeWitt. D, Turbyfill, C., Source code for Wisconsin Database Generator distributed on the "Wisconsin Benchmark Tape", Computer Science, U. Wisconsin, Madison, WI. 1984

[Bitton 2] Bitton, D., et al., "Benchmarking Database Systems: A Systematic Approach", 9th VLDB, Nov. 1983.

[Coppersmith] Coppersmith, D., et. al., "Discrete Logarithms in GF(p)", Algorithmica, **1**(1), 1986, pp. 1-15.

[Thekkath] Thekkath, C.A., Levey, H.M., Limits to Low-Latency Communication on High-Speed Networks, ACM TOCS, **11**(2), May 1993, pp. 179-203.

[DeWitt 1] DeWitt, D. et. al., "Gamma - A High Performance Dataflow Database Machine", Proc. 12th VLDB, Sept. 1986, pp. 228-236.

[DeWitt 2] DeWitt, D., et. al., "The Gamma Database Machine Project", IEEE Trans. on Knowledge and Data Engineering, March 1990.

[DeWitt 3] DeWitt, D.J., Naughton, J.F., Schneider, D.A. "Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting", Proc. First Int Conf. on Parallel and Distributed Info. Systems, IEEE Press, Jan. 1992, pp. 280-291.

[Englert] Englert, S., et. al. "A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases", Proc. of 1990 ACM Sigmetrics Conference. May 1990.

[Gerber] Gerber, R.H., "Dataflow Query Processing Using Multiprocessor Hash-Partitioned Algorithms", Ph.D. Thesis, Comp. Sci. TR 672, U. Wisconsin, Madison. Oct. 1986.

[hobbs] Hobbs, L., England, K.,*uren*, Digital Press, Bedford, MA, 1991.

[Horst] Horst, R., Chou, T., "The Hardware Architecture and Linear Expansion of Tandem NonStop Systems", Proc. 12th Int. Conf. Computer Architecture, June 1985.

[Jain] Jain, R., The Art of Computer Systems Performance Analysis, John Wiley & Sons, New York, 1991

[Kim] Kim, M.Y., "Synchronized Disk Interleaving", IEEE TOC, **3**(11), Nov. 1986. pp. 978-988.

[Knuth] Knuth, D., The *Art of Computer Programming,* **V**. 2, Chapter 3, 2nd ed., Addison Wesley, 1981

[Kronenberg] Kronenberg, N., H. Levey, W. Strecker and R. Merewood. “The VAXcluster Concept; An Overview of a Distributed System.” *Digital Technical Journal*. **1**(3): Jan. 1987, pp. 7-21.

[Nyberg] Nyberg, C., Barclay, T., Gray, J., Lomet, D., "AlphaSort - A High-Speed Sort for RISC Machines" Digital San Francisco Systems Center Technical Report TR 93.2. February, 1992.

[Press] Press, W.H., Teukolsky, S.A., Vettering, W.T., Flannery, B.P., *Numerical Recipes in C - 2'nd Edition*, Cambridge Univ. Press. Cambridge, 1992

[Ripley] Ripley, B.D., *Stochastic Simulation*, John Wiley, 1987.

[Schrage] Schrage, L.E., "A More Portable FORTRAN Random Number Generator", ACM TOMS, V. **5**(2), May 1979, pp 132-138.

[Smith] Smith, M., et al., "An Experiment on Response Time Scaleability", Proc. 6th Int. Workshop on Database Machines, June 1989

[Stonebraker] Stonebraker, M., "The Case for Shared-Nothing", Database Engineering, V. **9**(1), Jan. 1986.

[Tanenbaum] Tanenbaum, A.S., *Operating Systems: Design and Implementation,* Prentice Hall, 1986.

[Teradata] "The Genesis of a Database Computer: A Conversation with Jack Shemer and Phil Neches of Teradata Corporation", IEEE Computer, Nov. 1984. or *DBC/1012 Database Computer System Manual, Release 1.3*, C10-0001-01, Teradata Corp., Los Angeles, Feb. 1985.

[TPC] "Transaction Processing Performance Council Benchmark A", Chapter 3 of *Performance Handbook for Database and Transaction Processing Systems*, Morgan Kaufmann, San Mateo, 1993

[Uren] Uren, S., "Message System Performance Tests", *Tandem Systems Review*, V3.4, pp. 27-32, Dec. 1986.