Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds

ARRAY-DRIVEN SIMULATION OF REAL DATABASES

William S. Keezer

LEXIS-NEXIS
P.O. Box 933
Dayton, OH 45401, U.S.A.

ABSTRACT

A method to represent actual relational databases with arrays for simulation modeling of their performance as part of a software/hardware system has been created. The method includes a representation using arrays of the transactions accessing the database. This leads to a simplified, efficient submodel with a small network that gives accurate, detailed results from a detailed representation of the database without an increase in runtime or complexity of the overall model.

1 INTRODUCTION

In our work, modeling on-line transaction processing systems (OLTP's), response time and resource utilizations (CPU and disc) of transactions, as well as modeling groups of transactions with varying arrivals are important in assessing the success of software system designs and implementations. We have reported on our method of creating realistic workloads (Keezer, Fenic, and Nelson, 1992), our methods of supporting software development (McBeath and Keezer, 1993), and an efficient method of simulating UNIX disc I/O (Nelson, Keezer, and Schuppe, 1996). In this paper we report on our method for simulating relational databases(RDB's) and relational database managers (RDBM's), which can form an extremely significant part of the performance of an OLTP system.

In the last several years over thirty papers have appeared in which simulation was used to study relational database performance. The greatest number of these studies used simulation to evaluate some algorithm or approach to relational operations, e.g., the join process (Kitsuregaws, Harada, and Takagi, 1993), query processing (Pakzad, Jin, and Miller, 1991) (Fan, Su, 1993), and hardware configurations (Abdelguerfi and Sood, 1991) (Zhu, Han, and Hurson, 1992). A few papers used simulation to validate queuing analyses. Jhang, Kim, and Dean (1990) used simulation to evaluate the performance of relational algorithms on multicomputer architectures. Their purpose was to measure relative performance for defined queries. They explicitly did not model various workloads. Similar to our work, they parameterized the transactions though not with the methods we will present. To our knowledge, there have been no attempts to publish the simulation of actual databases with real workloads.

Our primary concerns are CPU, disc I/O resource utilizations, and overall response time. In our systems, memory has not been an important constraint. Since any RDBM will compete with the other transaction processing activities for the CPU and disc access, we must accurately and dynamically model these activities. Furthermore, since RDBM's often have many tables of various sizes, we want to be able to represent the workload in a realistic manner, varying accesses for different combinations of tables and operations on those tables. We also want be able to easily change table representations as database designers change their configurations.

Our first attempt to model a relational database was very specific to our application. The database parameters and processes in the model were hard-coded, which made reviewing and updating difficult. The CPU delay and I/O calculations were made throughout the simulation, wherever there were database transactions. This made the model very specific to our application and added overhead to the simulation. A more easily maintainable, more generic, and more structured model was needed.

This paper presents the important features of the method that was developed. The main features of this method are:

· Tables are represented as sets of parameters

· In-coming transactions are defined as sets of parameters which direct basic operations on different tables.

· Resource consumption is simulated by a small network of atomic operations, such as acquiring and utilizing CPU or performing disc accesses.

· Operations on tables are defined in terms of the atomic operations.

· The entire simulation of the relational database is isolated from the rest of the model, allowing it to be used as a submodel elsewhere.

The paper will first present an overview of the essential features of relational databases, followed by a presentation of the structure of the model. We will then present some results and discussion followed by a summary and conclusions.

2 IMPORTANT FEATURES OF RELATIONAL DATABASES

Three excellent sources for information on databases are Weiderholder (1977), James Martin's text (Martin, 1977), and C. J. Date's text (1991). Weiderholder deals mostly with non-relational databases and their performance. Martin and Date deal primarily with relational databases and their design and implementation.

In Chapter 1 of his book, C.J. Date (1991) describes relational systems as ones in which the user perceives the data as tables with operators that generate new tables from the old, as when one extracts data for a report. Some of the products that are relational in nature include DB2, SQL/DS, OS/400 Database Manager, RDB/VMS, ORACLE, INGRES, SYBASE, and INFORMIX. The differences in these products are primarily in the implementation of the actual atomic operations and minor details in the storage of the tables and memory management. For purposes of this paper, the important parts, the representation of the tables and their manipulation, are more similar than different among the products.

2.1 Index Structure

The key to accessing data as if it were in tables is the indexing of the values by which the data will be referenced. In RDB's a set of B-trees of indices to data is created. An index in this case is essentially a sorted list of the values of interest associated with the address of the record(s) containing that value.

In storing the data and indices, individual records are generally not stored, but collections of records called pages are stored. A page is a unit of memory and disc storage that may vary with the product, but is a power of two in size, e.g., 2048 or 4096 bytes. The capacity of a page is as many records as can fit intact on the page. The addresses of pages are kept in the indices. Both index pages and data pages have similar structures, the main difference is that an index record (or row) is not necessarily of the same length as a data record. Generally it is smaller, and normally more rows are stored on an index page than on a data page. There is always a single root page which forms the start of the B-tree There can be different numbers of levels in an index. There can be tables with only a root page, or tables with five or more levels of indexing. Generally, no more than five levels are implemented, because performance and storage considerations become paramount.

Whenever a table is accessed, the B-tree is read in several steps to find the correct location of the data. The first step is to read the root page and determine which of the pages it indexes contains the correct value range. This step is repeated for each level between the root and data pages. Then finally, read the data page. When a page at any level is read, it is scanned sequentially for the index or record of interest. In cases where the database is modified (adding, deleting, or modifying a record) the operation is recorded in a log (RDBlog) which is used to recover data in cases of failure.

There is one other concept necessary to our understanding of the storage structure and indexing, which is clustering vs. non-clustering. Clustered data is stored physically close together on the disc. For an employee record example, if the most frequent access is by employee ID, then the records should be stored on disc in either ascending or descending employee ID order, and the employee ID index would be said to be clustered. Any indices to other data values would be unclustered, in that the sequential indices in their pages would point to data pages that are spread over the physical disc space in a more or less random fashion.

Database designs are compromises (Date,1991). A design that provides efficient updates may not allow rapid and flexible queries. Queries are fastest when most of the data requested are indexed, requiring multiple indices for every record added. If many records are added in an update, this overhead can drastically effect the performance of the system. On the other hand, if updates are fast and efficient, the queries may suffer from lack of indexing and large numbers of sequential reads of data may be required to obtain the desired information.

2.2 Stored Transactions

Data in RDB's is accessed by a language called SQL or a variant thereof. Any one access requires a series of SQL statements which perform the necessary operations to retrieve, insert, remove, or modify the data. It is the SQL which provides the view to the user that the data is in tables though it is actually stored as indexed records. One of the advantages of this language is that statements can be grouped into procedures and stored as part of the database. Furthermore, once the procedures are established and the database populated, the procedures may be optimized and compiled in their optimized form and then called during runtime, similar to a subroutine. In this manner, an incoming transaction only has to call the proper procedure to accomplish all its desired results.

2.3 Operations On Data

For the purposes of our models, we found that all of the SQL commands used in our stored procedures could be implemented with five operations, FETCH, STORE, REMOVE, CHANGE, and COMBINE. The FETCH operation does just as it sounds; it finds records by descending the index tree, starting at the root level. We use this operation as the beginning of all other operations, and as a substitute for such SQL statements as SELECT or IF EXISTS.

The STORE operation reads down the tree to find the correct location for the record operation, and then adds a record. After the addition of a record there is further work, in that there may not be space in the data page for the new record. In such a case, a new page is created and half the data from the old page is moved to the new page, and the record is inserted on the correct page. The indices for the old page must be updated to reflect the change in value range, and a new index to the new page must be inserted in the index page. This in turn may cause further addition of index pages at the next higher level, potentially on up to the root. In the case of a split of the root page, another level is added to the B-tree. We do not model the addition of a level. There is a variation on the STORE operation when records come into the system in sequential order of their clustered index. In such a case, the system would not create a new page until the current page is full, and would not move data to the new page, but would only put the new record in the first slot. The remaining index maintenance is the same.

The REMOVE operation also reads down the index tree to find the correct location. The record is then removed from the data page and the following records are moved up. If the entire page is empty it would be deleted and its index deleted at the next higher level. The combining of partially-filled pages and the deletion of empty pages commonly occurs only during a restructure of the database at a low-usage hour, and would not be important in the performance considerations.

The CHANGE operation is very similar to FETCH if the data is in a fixed format. The index tree is read to the correct location of the record and the correct field changed. If, however, the data is in variable format, the old record is deleted and the new record inserted.

The COMBINE operation is used when data from two or more tables is to be extracted and joined to form a new table. If the keys are be the same, we would model such an operation not as a COMBINE in our sense, but as two FETCHES with extra processing to do the data merge.

If the two tables have no common keys, for each row on one table, every row in the other table would have to be read to extract the relevant records. In our models, this is the COMBINE operation, referred to in the literature as an outer join. This operation has major performance implications. [There are a number of other implementations of this operation. For an excellent review see Mishra and Eich (1992).]

3 STRUCTURE OF THE SIMULATION MODEL

The model contains four major parts, the representations of the transactions, the representations of the tables, the subroutines for calculating resources, and the simulation network itself. This section starts with a high level description of the overall simulation process, then follows with more detailed descriptions of the various parts of the model.

3.1 Process Description

A transaction is sent to the relational database, which triggers a stored procedure. This stored procedure accomplishes the desired access to the data as a combination of SQL logic steps and table operations. The results are returned to the requester. In our model, the purposes of the RDB model were to properly compete for resources and to expend clock time in the approximately correct amounts, using a small network. Though we returned no results to the model, it would be possible to do so.

The flow in the model is based on table operations, stepping through them sequentially. When a request arrives, the model finds the correct stored procedure parameters, loads the first table's descriptive parameters, and loads the transaction's parameters for that operation. The resource consumption for the table operation is calculated in a subroutine, and the resources are expended in the simulation network. When that operation is finished, the entity returns to the stored procedure and obtains the next table operation and repeats the process. This cycling continues until all the table operations are completed. When the stored procedure is completed, the entity is returned to the requesting part of the simulation.

3.2 Representing Transactions

SQL transactions are viewed as being a series of table operations. Each operation has five parameters, the primary table identifier, the secondary table identifier (if the operation is a COMBINE), the number of records to be accessed in the primary table, the number of instructions, other than table operations, in the SQL procedure to be executed, and the operation identifier. If the number of records to be accessed is a variable for this step in the procedure, then this value can be a negative number pointing to the location of the correct number of iterations. Since one procedure can consist of multiple table operations, each SQL transaction is represented by an $(N + 1)$ by 5 array, where

N represents the number of tables to be accessed, and five is a column of the five parameters for a given table operation. The additional column is used to provide a negative one as the table identifier, indicating that the procedure is completed.

Ad hoc inquiries, in principal could be provided for by generating the necessary parameters as required and placing them in an array built for that purpose.

3.3 Representing Relational Tables

Tables were parameterized with a set of sixteen values, two of which were left unused for possible later expansion.

The following is a list of the Table Definitions in the model storage array.

1. Table Identifier
2. Number of pointers in the root, zero if a flat file
3. Number of pointers in the rightmost page of the first branch
4. Number of pointers in the rightmost page of the second branch
5. Number of pointers in the rightmost page of the third branch
6. Number of pointers in the rightmost leaf
7. Capacity of the first branch
8. Capacity of the second branch
9. Capacity of the third branch
10. Capacity of the leaf
11. First branch contents after a split; if zero, the keys are sequential and pages are added, not split. If negative, the steady state has equal numbers of deletions and additions and few splits
12. Second branch contents after a split; if zero add, if negative insert but don't split
13. Third branch contents after a split; if zero add, if negative insert but don't split
14. Leaf contents after a split; if zero add, if negative insert but don't split

15-16. Left for expansion

3.4 Calculating Resources Used

The processing for a FETCH operation determines how many pages are to be read from disc, and how many pages total are read by the process To determine the number of pages read in processing, the subroutine steps through the current capacity values, from root to leaf (the lowest level in the index tree), and counts the non-zero levels. The number of pages to be read from disc equals the number of levels above two. (We assume that the root and the first level are always memory-resident, and all higher levels must be read from disc). CPU consumption is based on the number of instructions executed for each step in addition to table operations.

STORE processing follows the same logic as FETCH to locate the location of the new record. It then calculates how many pages are split or added, and how many rows or pointers are inserted or added. The subroutine then looks at the current leaf contents. If it is not equal to the capacity, it adds or inserts a pointer, based on whether the split value is non-zero (insert) or zero (add). If the page is at capacity, it is split or a page is added, and the next level up is processed in the same manner with the pointer to the new page. This can cascade to the root level of the tree. However, this model does not split the root, since this would create a new level for the table index tree.

CHANGE follows the FETCH logic to find the location for the update. It then calculates the modification process as a pointer or row addition, and then exits. Updates are assumed not to create splits or adds, even though there may be variable length fields involved.

REMOVE is identical to STORE with the signs changed for the page content updates. The logic is also reversed for determining page removal. If the page is empty, a pointer is removed from next level, but a disc write for the empty page is not charged.

COMBINE has a totally different process from the other four operations. RDB's do joins in a stepwise manner, if more than two tables are involved. It nests the tables in a hierarchy based on their size at the time the procedure was compiled, the smallest innermost and the largest outermost. RDB then steps through ALL rows of the innermost table once for EACH row of the next table out, and creates a temporary table for that result. It then takes the next table out and steps through it, scanning all rows of the temporary table once for each row in the outer table, as before. This process continues until all tables have been joined.

In modeling the COMBINE process, the model assumes that any temporary tables will have the same number of rows as the outer table of the join that created it. The number of rows scanned is the total number of rows in the inner table multiplied by the number of rows in the outer table. This would be a worst case, as there may be constraints on which rows of the outer table are joined. They all have to be read, however. For the process, the outer table is the Table ID in row one of the procedure set in the storage array, and the inner table is the Inner Table ID in row two.

The next calculations are performed for each table in the JOIN. The first calculation made is the number of leaves in each table. This is the Π-product of the current contents of the root and all branches. The sum of the number of leaves becomes the number of pages to read from disc for tables with leaves in levels 3, 4 or 5. The next calculation is the number of leaves times the number of rows per leaf. For tables that are split to add leaves, use the average fill factor of 75% or assume the current contents are representative (pessimistic for >75% and optimistic for <75%). For tables that add leaves, calculate the number of leaves less one, then multiply it times the capacity of a leaf, then add the current capacity to that. The sum of the rows for each of the tables is the number of rows scanned.

The remaining calculation is the number of pages to be written to disc for the final table. Assume a generic row size, multiply that times the number of rows in the outer table, and divide by 2K or 4K depending on page size. Since join operations are generally used to create tables for output to printers or CRT's, 200 bytes/row or less would be a reasonable number. Using a value of 200 bytes and a page size of 2K would lead to about 10 rows per page. Since there is some overhead per page, simply dividing the number of rows by 10 would provide a reasonable estimate of the number of pages in the resulting table from a COMBINE.

3.5 Simulating the Consumption of Resources

There are nine values necessary to simulate the resource consumption of table operations. Not all of them are used in every operation. The values are the RDBlog entry size, the number of instructions executed in the stored process, the number of pages to be read from disc, the total number of pages read (memory and disc) for processing, the number of pages split, the number of pages added (not part of the split, used in sequential key table), the number of rows/pointers modified/added, and the number of rows/pointers inserted, and the number of rows scanned. These resources are consumed in a small sub-network described in the pseudocode which follows. This network would vary with the various RDBM products, since it closely reflects their implementations.

```
Obtain the RDB manager
Obtain a CPU
Consume CPU to setup the operation
Release the manager
        (Disc I/O is under the operating system)
If disc reads are required to obtain data
  While disc reads are >0
  Execute single appropriate disc I/O simulation.
  Decrement disc reads required
     End While
     End If
Obtain the RDB manager
Wait for CPU
Consume CPU for number of instructions executed
Consume CPU for pages read
Consume CPU for rows scanned
If pages are added (zero value for pages split)
  Consume CPU for pointers added
  Consume CPU for pages added
     Else
  Consume CPU for pages split
  Consume CPU for pages inserted
     End If
If data has been modified
                (STORE, CHANGE, DELETE)
     Increment RDBlog contents by the RDBlog entry size.
  Calculate the number of log pages to write.
  Calculate the number of disc writes.
  Add one write per log page.
If a page is split
     Writes are added equal to the twice number of pages split
     Add the number of inserted pointers less the number of pages split
If the page is not split
     Writes are added equal to the number of pages added
     Add the number of pointers added less the number of pages added
Execute the number of disc writes calculated.
Return to the correct operation routine.
```

There are two daemons to be simulated in the RDB sub-model. The RDBlog daemon writes one or more RDBlog pages whenever the criteria of the RDBM are met. The memory daemon flushes pages to disc whenever the RDBM criteria are met. The concept for both daemons is to detect when writes are needed, then queue the necessary number of page-requests as write I/Os.

4 CREATING PARAMETERS

4.1 Defining the Stored Procedures

Stored procedures consist of executable statements and table operations. The table operations in turn require CPU and disc resources. The monitors for the RDBM may give information on how these procedures are compiled and optimized. For example, in SYBASE one can use SET PLAN to find the order of execution of statements and the exact order of table operations. SET IO gives the number of real I/O’s and logical I/O’s (memory reads) occurring during table operations. There are also monitors which can give elapsed times and CPU utilizations.

One must be careful in interpreting these data, especially if determined on databases under development. The listing of the compiled procedures may be difficult, as they are not executed in the exact order listed in the SQL code; called subroutines execute immediately after the calling instruction. This nesting of execution steps can lead to confusion, if not carefully followed. There may be more logical I/O than will occur in production, or there can be access paths, which are optimal for the partly populated database, but are very inefficient for the full database. The monitoring process adds to both the CPU and the elapsed time, and these values should be considered upper bounds.

From these data, the number of instructions executed and the order and type of table operations can be placed in the stored procedure vectors. The number of real vs. logical I/O's is determined by assuming only the root and the first level of indexing are in memory. This assumption has appeared to be almost universally applicable in the author's experience, because fan-out at the third level of indexing creates far too many pages to fit a significant number of them in memory. The CPU utilizations were determined by benchmarks.

4.2 Defining the Data Tables

The performance of an RDB is the sum of the atomic operations necessary to access the various tables. These depend on the number of levels in the tables, how many rows of indices there are on each page, and how many pages are memory resident. This is obtained from the database designers, but generally treating each table separately will be too unwieldy because of the large number of tables. Frequently there will be groups of tables with similar structure for indices and data pages, and there may also be tables that will not be used at all during the period being simulated. For example, one transaction we modeled had 10 different tables. One accounted for over 65% of the accesses. The another 30% of the accesses were to tables with similar key structure and levels of indexing, so that the transaction was simulated with only two modeled tables.

The most important parameter is the number of levels in the B-tree. It is not possible to simulate a three-level table with a two- or four-level table because every level over two requires a disc read. Modeling a three-level table with a two-level table will use insufficient resources and modeling it with a four-level table will use too many.

The next parameter is whether the index is primary or secondary. Primary indices generally have the data pages clustered with them and have a different leaf (bottom level) structure than secondary indices, even if the number of levels is identical. Primary indices may be inserted or added depending on data clustering and sequentially of keys. Secondary indices are always inserted. One may combine similar secondary indices into one table model, but not primary and secondary.

The third parameter has the most room for combining tables. It is the number of index rows in the root and branches and the number of data or index rows in the leaves (depending on whether it is a primary or secondary index tree). If the number of levels are the same, there is little difference in the work to scan half a page, e.g. the difference between scanning 30 and 40 entries. In our work we had a number of tables with three levels and between 40 and 48 index rows per page. Since the data rows were similar in size, the leaves were similar, and all were represented as one table. The same would hold true for secondary index table leaves.

4.3 Calibrating the Performance

Calibration of the model is of course critical to its success in predicting resource consumption. The values requiring calibration were for CPU consumption by various table operations and for disc I/Os.

When we first created the model, the disc I/O values at the hardware level were obtained from manufacturer's data for average seek, latency, and data transfer rates. Since that time, a generalized model for I/O has been developed that simulates each individual I/O (Nelson, Keezer, and Schuppe, 1996). If the RDBM does its own disc accesses, an estimate or a measurement of the CPU time to handle the I/O's is required. Where the RDBM utilizes the operating system to access disc, an estimate or measure of the CPU time to generate the request is required. There will also be the expenditure of CPU by the operating system for each I/O. This value would be whatever is used in the rest of the model.

We were faced with the challenge of calibrating the CPU consumption for the table operations. For this we used a custom benchmark consisting of a series of record storage operations with many repetitions and with the number of records varying over an order of magnitude. We plotted the response time against the number of records. The slope of the line indicated the storage time per record. We found that there was little or no queuing time in the elapsed time and that to a good approximation, it was all CPU time.

We then used this CPU time value to determine the cost of table operations. There were seven values of interest: row read, page read, row addition, page addition, instruction execution, page splitting, and row insertion. With one CPU value and seven unknowns, we took reading a row of data as the smallest operation and built upon it. We chose to equate reading a page header in memory and processing its data (page read) to reading five rows of data. We set adding a row of data to the end of a page as equivalent to 10 row-reads. This helped account for the RDB log work that an add operation generates. We then made adding a new page to a table the equivalent of five page reads to cover the overhead of creating and allocating a new page image in memory. Additionally, in random tables splitting a page added a further 2.5 times the page add cost, because half the data must be copied to the new page. Inserting a row in a page had an additional 2.5 row read cost to cover the cost of pulling down the rows on the page that would follow the new rows. Finally, execution of a stored instruction that had already been compiled was taken as twice the cost of reading a row.

By determining the operations required to add a record to the database, and then equating that to row-reads, one can find the total number of equivalent row-reads. This is set to the CPU cost of adding a record, and the CPU cost per row-read is calculated. One can then back-calculate to find the costs of the other operations. We found that it took the equivalent of 181.4 rows read in 7.8 msec, giving a CPU cost of 0.043 CPU msec per row read. After the back-calculations to find the other six values, the only operation that had not been calibrated was the row delete. This was taken to require the same cpu costs as an add. It is important to note that the particular ratios we used worked well for us, but may not work for all systems and all RDBM's. The general approach, however, would work for all systems and could be refined depending on the available metrics for calibration.

Later comparisons of the CPU utilizations of the RDB system estimated by this model to actual tests of implemented code showed the model estimates were within 3-5% of the total system CPU actually utilized, e.g., the model predicted 7-9% of system CPU for the RDB and the actual utilizations were 7-11%.

5 RESULTS AND DISCUSSION

The method we developed to parameterize relational databases has a number of benefits. One is that the generated data and calibration of operations has allowed the estimation of the performance of databases with spreadsheets in many instances, allowing a sufficient estimate to be made without developing and running a model. In those cases where a simulation is required, it is relatively easy to change the database simulation to reflect the new or different configuration. To define completely the structural parameters of the relational database and the operating parameters of the database manager requires a fairly large effort. However, at early stages of the design of the database, generic place-holder values can be used, the values of which are estimated from past experience.

The structure of the model was quite different from our earlier efforts. We consumed all the resources in a small network which was executed after the calculation of the resources to be consumed. These calculations were performed in compiled user subroutines, e.g. the EVENT node or USERF in SLAMSYSTEM®. The remainder of the network in the RDB model was mainly routing of entities to the correct procedural calls. In addition to parameterizing the database structure and the operations, we also parameterized the stored procedures for the incoming transactions. Thus an incoming transaction was linked to a set of vectors that defined the sequence of operations and tables to be manipulated. As a result, the maintenance of the transactions, the database structure, and the relational manager simulations were entirely separate. As can be imagined, this greatly simplified the overall maintenance of the RDB model.

Even though we modeled the RDB in much more detail than previously, the smaller number of nodes, and the reuse of attribute space in the sub-model prevented the run-time of the overall simulation from increasing. Furthermore, the accuracy as mentioned earlier was exceptionally good. Part of the increased accuracy was due to the deliberate choice to model the measured elapsed times in the calibration runs as if they were all due to CPU time. This provided a compensation for the use of deterministic values in the model and the lack of explicit modeling of the operating system background functions.

One new use of match nodes in SLAMSYSTEM came from this effort. We used the match node to isolate the sub-model from the rest of the system in order to reuse the attribute space. In cases where the results of the sub-model are desired but further processing will occur to the original entity, we would recommend this over a straight hierachical addition of submodels since in order to preserve attribute values, it would be quite possible to explode the attribute space with a negative impact on run-time.

6 SUMMARY AND CONCLUSIONS

We have described a method for modeling relational databases that is flexible, accurate, and efficient in its use of resources. This method may be used to model various relational databases without changing the simulation network, and can model different relational database managers with changes to only a small portion of the network. The process of gathering the data and calibrating the model enables static estimates of performance using spreadsheet techniques, where a model run is not justified. Additionally, we have created a novel use for the SLAMSYSTEM match node as a means of subnetwork isolation, allowing complete freedom in reusing attributes of entities. Though our purpose has been to simulate the performance of relational databases, there is no reason these techniques cannot be used for any indexed database with appropriate modifications.

ACKNOWLEDGEMENT

Great appreciation is expressed to former colleague and one-time co-author, Darby F. McBeath. Her comments during the writing of this paper, and the heroic efforts she exerted during the building of models have been a tremendous asset.

REFERENCES

Abdelguerfi, M. and Sood, A.K. 1991. A fine-grain architecture for relational database aggregation operations. *IEEE Micro 11*, 6, 35-43.

Date, C. J. 1991. An Introduction to Database Systems, Vol. 1. 5th ed. New York, New York: Addison-Wesley Publishing Co.

Fan, J.-J.., and Su, K.-Y. 1993. The effect of database filters on the performance of bufferred relational database systems. *Information Systems 18*, 2, 99-109.

Jhang, H., Kim, T. G., and Dean, R. H. 1990. Modelling relational databases on multicomputer architectures. *Proceedings of the 1990 Summer Computer Simulation Conference*, ed. B. Svrcek and J. McRae, Society for Computer Simulation.

Keezer, W.S., Fenic, A.P. and Nelson, B.L. 1992. Representation of User Transaction Processing Behavior with a State Transition Matrix, *Proceedings of the 1992 Winter Simulation Conference*, ed. J.J. Swain, D. Goldsman, R.C. Crain, and J.R. Wilson, 1223. IEEE, Piscataway, NJ, 1992. p. 1223

Kitsuregawa, M., Harada, L., and Takagi, M. 1993. Algorithms and performance evaluation of join processing on KD-tree indexed relations. *Transactions of the Institute of Electronics Information and Communication Engineers D-I J76D-I*, 4, 172-183.

Martin, J. 1977._*Computer Data Base Organization*. 2nd ed. Englewood Cliffs, NJ: Prentice-Hall.

Mishra, P. and Eich, M.H. 1992. Join processing in relational databases. *ACM Computing Surveys 24*: no.1, 63-113.

McBeath, D.L., and Keezer, W.S. 1993. Simulation in Support of Software Development. *Proceedings of the 1993 Winter Simulation Conference*, ed. G.W. Evans, M. Mollaghasemi, E.C. Russell, and W.E. Biles, 1143. IEEE, Piscataway, NJ, 1993. p. 1143.

Nelson, B. L., Keezer, W. S., and Schuppe, T. F. 1996. A Hybrid Simulation-Queueing Module for Modeling UNIX I/O in Performance Analysis. *Proceedings of the 1996 Winter Simulation Conference*, ed. J.M. Charnes, D.M. Morrice, D. T. Brunner, and J.J. Swain, 1238. IEEE, Piscataway, NJ: p. 1238.

Omiecinski, E., Liu, W., and Akyildiz, I. 1991. Analysis of a deferred and incremental update strategy for secondary indexes. *Information Systems 16*: no. 3, 345-356.

Pakzad, S. Jin, B.. and Miller, L.L. 1991. Design and analysis of an intelligent support system for large databases. *Proceedings of the Twenty-Fifth Hawaii Intgernational Conference on System Sciences 1*: 348-58.

Weiderholder, G. 1977. *Database Design*, New York, NY: McGraw-Hill Book Co.

Zhu, J., Han, J.Y., and Hurson, A. R. 1992. A multiprocessor organization for very large relational databases. *Proceedings of the 34th Midwest Symposium on Circuits and Systems 2*: 970-973.

AUTHOR BIOGRAPHY

WILLIAM S. KEEZER is currently a Senior Systems Engineer focusing on mainframe storage management and has been with LEXIS-NEXIS for over eleven years. He holds B.S. and Ph.D. degrees from the University of Oklahoma, and is a member of the ACM.