Proceedings of the 1986 Winter Simulation Conference
J. Wilson, J. Henriksen, S. Roberts (eds.)

DISTRIBUTED DATABASE QUERY SIMULATOR

Raju Kocharekar
The World Bank
Washington, DC 20007

ABSTRACT

This paper discusses a design methodology for a Distributed Database Query Simulator (DDQS). A distributed database query consists of two different types of sub tasks; One, the tasks pertaining to the relational database manipulation language, such as project and joins on different files, and two, the tasks of file transfers. A given query can have any number of subtasks of both the types. These tasks are interrelated to each other, and parallel operations can be performed provided the required serialization is maintained. The subtasks compete for common resources with subtasks from other queries being evaluated at the same time, or to some degree, with the subtasks from the same query, if performed parallel. The purpose of the DDQS is to evaluate the performance of a given query algorithm in terms of the total query processing time, in a given workload and network architecture. The DDQS uses transaction oriented GPSS language to simulate the query, while pascal is used to generate the GPSS source code relevent to the query. The DDQS design is highly modular and simulation of any communication protocol layers or database access methods can be changed without affecting the higher level modules.

1. INTRODUCTION

1.1 What Is A Distributed Database Query?

In a relational database system, users can extract the information from the database in a non procedural relational calculus language, or a procedural relational algebraic language, or a hybrid of the both. In accessing the data the user conceptually views the database in a tabular form, irrespective of the individual file structures. The same concept is even further generalized in the distributed systems, by accessing the data from different locations without the user awareness. A database dictionary is used for the mapping of logical and physical structure and location of the data files. The file locations are decided statically at the database design time, or dynamically based on the statistics collected on the database queries for file accesses.

1.2 Distributed Query Processing Algorithm

A distributed database query processing algorithm is a procedure of tasks or operations, which if and when executed, results in the output data fulfilling the user requirement. There are two main types of tasks in the query processing algorithm.

i. The Relational Data Manipulation language (DML) tasks
ii. The file transfer tasks

The relational data manipulation language tasks perform relational operations on one or more database files generating a unique ouput file. For example, a 'select' relational operation performed on an employee file with the selection criteria of 'Employee LastName = Smith' will result in a database output file with all employees with lastname 'Smith'. A 'join' relational operation performed on the employee file and the department file on the department number (where both the files have department number as a field in the record formats), will result in a file with each employee record having information about the department of the employee, such as the department name.

Note that most of the relational DML tasks are similar for the centralized as well as distributed data base system, except for the few, such as 'semijoin' or 'union' which are especially beneficial in case of distributed database systems. The reader is requested to refer to Date (1986) for further details.

The output of a DML operation is a relational table, similar to the original relational database tables used in the operation. Depending on the query processing algorithm, the output can be used in the subsequent processing, or can be directly taken as the result of the query, if this DML operation is the last step in the algorithm.

The file transfer tasks are integral part of the distributed systems. In the beginning, or at any later point in the query, a data file must be accessed at a remote location to process the query. For example, in the bank database, information pertaining to a particular customer is most likely to be found at the branch location, where the customer has the account, and must be transferred over the communication lines if required at any other locations.

Along with the data file transfers, a distributed data base system also incorporates the control message transfers. The control messages are instructions to the receiving site. In the banking database example, a message must be sent to the customer account site to request for the customer data. These messages, in general utilize the same communication lines as the regular data files.

1.3 Query Optimization

A typical user query can be fulfilled in many different ways. Two different criterias are used to evaluate the query performance; the cost of the query and the response time. Many methods are proposed in the literature to find the optimal way of processing the query, based on either the cost or the response time, with an assumption of proportionality between the two measurements. Unfortunately, finding an absolute optimum algorithm of query processing is a computationally explosive problem (n-p complete). The solutions therefore are heuristic.

The query processing costs can be split into two types of costs, mainly the DML processing costs, and the communication costs. The two costs can have various proportion of the total cost, depending on the network architecture and the database management system used. These costs are dependent on the total execution time of the individual tasks, but not on the time the task is performed. e.g. the cost of a join of two relational files depends on the total processing time required for the operation, irrespective of when the join is performed. Similarly, the cost of sending a file depends on the size of the file, and not on at what time the file is sent.

The total response time for the query depends not only on the execution time required for each individual task, but also on the time of execution. The overall response time of the query executed by concurrantly running many different tasks is lower than the response time of the query executed serially. It is essential, however to know that not all the tasks in the query alogorithm can be executed in parallel. The syncronization needed in performing these parallel operations is explicit in the given query processing algorithm, and has to be maintained in the execution of the query. The methods to find the minimal response time query algorithms try to maximize the number of parallel tasks and minimize the individual task execution times.

1.4 Simulation Of A Distributed Database Query

It is possible to measure the cost of the query analytically, but it may not be easy to determine the response time, especially under different varying workloads. The analytical methods to calculate the response time may even be misleading under low workload. A query processing algorithm executes a few DML operations in parallel, and transfers the resultant files for the preparation of next DML operations at subsequent host nodes. The cycle is repeated until the final datafile is available at the user query initiation site. The host processors in turn are busy at certain time, while communication lines are relatively idle, and vice versa. It is incorrect therefore to calculate the host processor or communication line access time based on the overall average wait time, in a low workload. A Distributed Database Query Simulator (DDQS) can be effectively used to determine the response time under varying workloads. The DDQS also determines the utilizations of the underlying resources. The power of the DDQS lies in its use in determining the query responses under varying network and host computer architetures, when the initial architecture of the distributed base is designed, or when new resources are to be established, or the current ones to be relinquished.

Distributed database queries can be broken up into a few number of classes, depending on their nature such as the frequency, location, and the underlying processing algorithm. These classes of queries can then be simulated combined, or the few interested classes can be simulated against the background workload. The advantage here is to simplify the model by grouping the queries.

The DDQS has to be flexible enough to test under different database and network architectures, and yet specific enough to simulate any query algorithm. This requirement necessitates a moduler design of the simulator. The DDQS follows the ISO/OSI network model. In the ISO/OSI network model, distributed database query processing falls in the topmost application layer. To simulate the top layer, the inner layers of the ISO/OSI model have to be simulated to required extents. One should also be able to change the inner layer simulations without changing the interface to the higher layers.

2. DDQS DESIGN

2.1 Distributed Database Query Representation

A distributed query algorithm can be represented in many intermediate data structures, before finally converting into an output 'object code'; but the most commonly used format is a tree representation of the query. In this representation, leaf nodes represent the original data base files, while inner nodes represent the DML operations to be performed. The nodes at the bottom-most inner layers are executed first. Each tree node produces an ouput file to be used by its parent node. The root node produces the result. In some cases, it is also possible that the intermediate representation is an acyclic graph, indicating that the same file is used for two different DML operations. These cases are beyond the scope of this paper.

The file transfer operations are not explicitely represented in the tree structure, but the location in the network the DML operation is to be performed, is stored at the tree nodes. The resultant file characteristics are also stored at the tree nodes. A pascal data type for a tree node is shown in Figure 1.

```
DMLtasktype = (join, semijoin, ... ,
                                project);

attributetype = record
  attr_name : array [1..10] of char;
  attr_mode : (integer,real..char);
  nextattr_ptr : ~attributetype;
              end;

filetype = record
  filenumber : integer;
  filesize   : integer;
  attribute  : attributetype;
  end;

treenodetype = record
  DMLtask    : DMLtasktype;
  Leftson    : ~treenodetype;
  rightson   : ~treenodetype;
  location   : integer;
  ouputfile  : filetype;
  end;
```

Figure 1: Tree Node Data Type

It should be noted here that all the query optimizing algorithms try to estimate the DML processing required at each tree node, as well as the intermediate file sizes. This information therefore is available in the query tree representation, and is of vital importance not only to the DDQS for the simulation purpose, but also to the actual execution process for scheduling the resources. The actual conversion of the intermediate query to the 'object code' is done by traversing the tree, and generating the relevent code as each node is visited at some point during the query. This syntax directed translation is very much alike translation in the language compilers and the reader is reffered to any compiler textbook, such as Aho and Ullman(1977). The ouput query object code for the distributed database is a set of executable programs equal in number to the nodes involved in the query processing.

The query is executed in two phases. In the query initiation phase, the query processing programs are sent to the respective nodes over the communication lines (except for the query initiation node program). In the query execution phase, the programs are bootstrapped and executed at node sites. The programs consist of the database access primitives corresponding to the DML tasks, and communication access primitives pertaining to the file transfer tasks. Syncronization among the subtasks is achieved by establishing critical regions within the host node programs, or by simply waiting for the particular resources, which in these cases are relational files. For example, if a query processing program at a network node site needs a relational file for the 'join' operation to be performed, and the file is to be sent from a remote host node in the network, but has not arrived yet, the program waits until the file is received. In the reverse case, if the file has been sent from the remote host node even before the query processing node program has reached to the point in the execution where the file is utilized (join in this case), the file is received at the network node site and stored temporarily.

2.2 DDQS Design Methodology

The distributed queries are executed as parallel asynchronous tasks as described earlier. This asynchronous nature of tasks is dependent not only on the type of the query asked, but also on the query optimization method used. Since many algorithms are possible to process the same query, the query transactions process can not be hardcoded in the simulation model.

Unfortunately, the existing simulation languages can not handle complex data structures like trees effectively. An attempt has been made to store the entire query algorithm in the GPSS transaction parameters in Wnek and Roth(1984) in a linked list form. However, the model allowed only serial execution of the query processing through the linked list form. The model later was modified to allow for parallel transaction processing in Wnek(1985), but these transactions have to be initiated at the same time, and the processing could not be continued until all the generated parallel transactions are complete. In the intermediate tree structure for the query processing algorithm, only the parent or ancsestor tree nodes operations have to wait for the operations of the children nodes to be complete, and not the others.

An alternate method could be used to store the query algorithm tree in GPSS matrix savevalue form. The matrix representaion of a tree consists of rows for each tree node, (DML operation), with columns indicating the node number, its left and right sons, its parent node and the other specific information, such as the the kind of the DML operation at the node. The nodes with no left or right son have zero values in the respective columns. Similarly the root node has a zero value in the parent column.

In the matrix savevalue representation, seperate GPSS transactions are generated for each row for which no left or right son exists. At the end of any transaction the matrix row is accessed to find the parent node of the transaction. A zero value is placed in the corresponding son column entry at the parent transaction node row. The parent transaction is initiated if both left and right son entries are zero. The query processing is complete when the transaction with no parent tree node is executed.

The above approach is simple and easy to understand. However there are two drawbacks in the model. First, the GPSS transaction processing in this case is still data driven, and the model is difficult to debug for programming or data input errors. The second important drawback is that the algorithm uses the matrix savevalues, which are global in nature. Only one query processing transaction can be simulated at a time in this method because the matrix values can not be changed by more than one queries. This defeats the earlier mentioned purpose of grouping the queries into classes, and modelling the class of queries.

A more elegant approach to the design (Figure 2) is to generate a GPSS program source code specific to the query, rather than encoding the query algorithm in the data values. This is possible because the database queries are grouped into few classes and the generated source code is limited in size. Since the DDQS structure is highly modular, most of the lower level subroutines are copied from the libraries based on the DBMS and network architecures used, while only the top level block statements are generated from the query algorithm, in the source code assembly phase. By storing the communication and database access method protocols in the library, the library can be enriched with as many protocol subroutines as required.

The tree query data representation is used as an

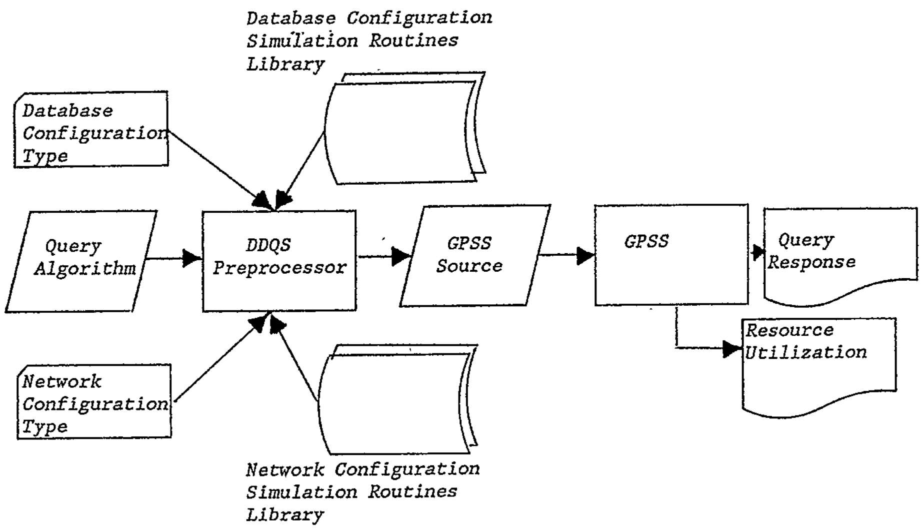


Figure 2: DDQS System Flow-Chart

input in the source generation. The tree is traversed the same way it would be traversed in the query object code generation. In fact, the query object code can directly be used instead of the tree representation, thus eliminating the duplication of work. However, the database access primitives as well

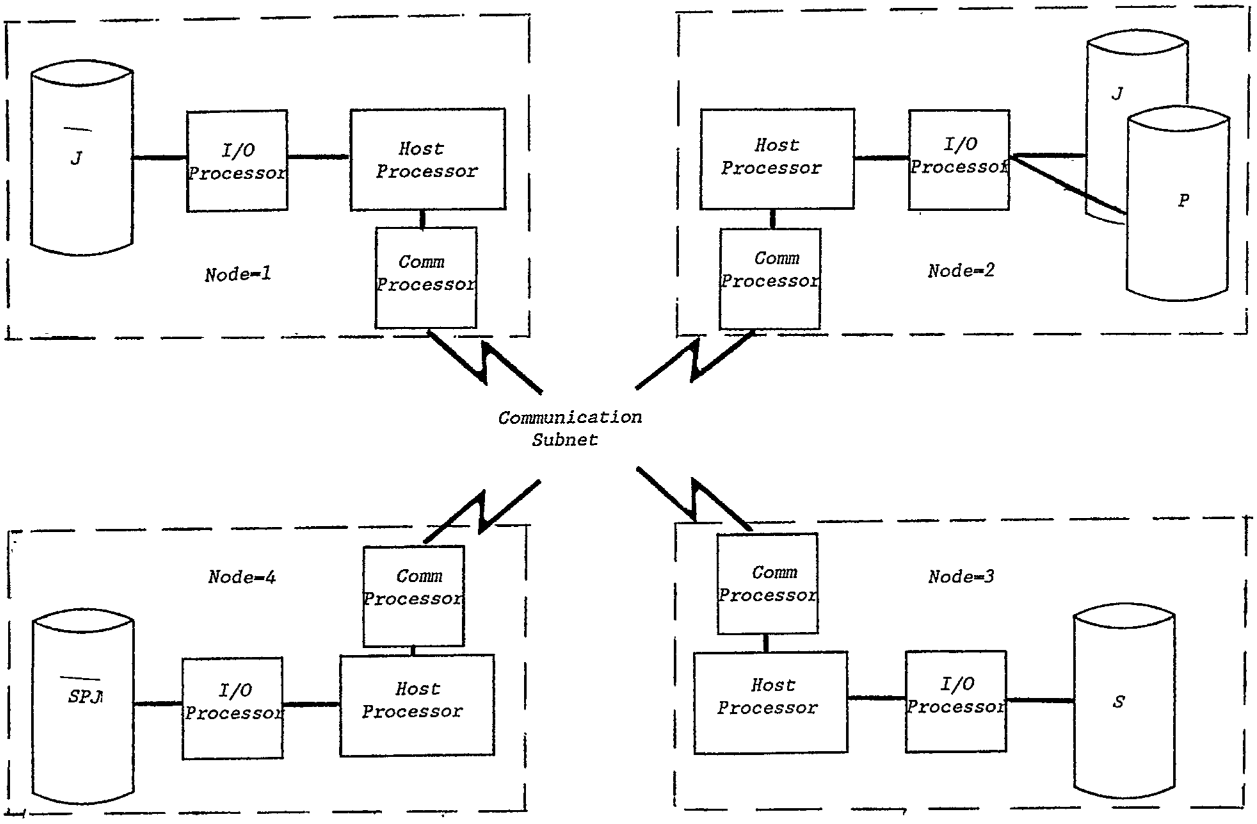


Figure 3: Distributed Database Network Configuration

as the communication access primitives in the query object codes are not standardized.

3. DDQS Implementation

3.1 A Sample Query

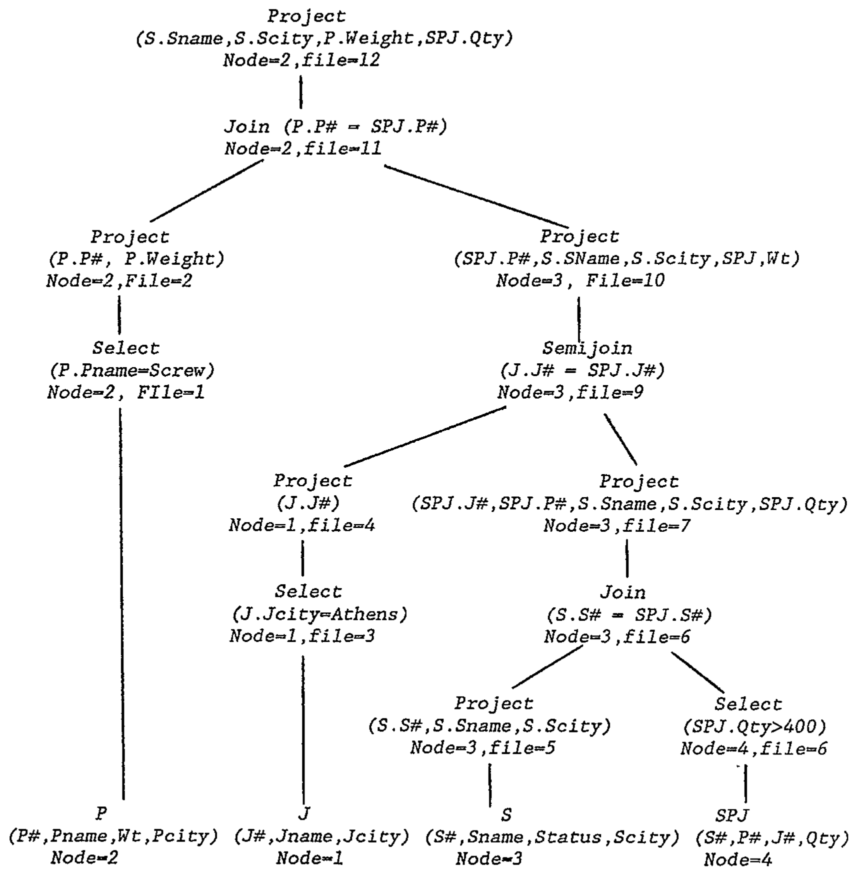
Consider a relational database of Suppliers, Parts, jobs as follows.

S	S#, Sname, Scity, Status
P	P#,Pname,Weight,Pcity
J	J#,Jname,Jcity
SPJ	S#,P#,J#,Quantity

The database has three entity relation files pertaining to above entities, and an association relation file describing quantities of parts required for each job and supplied by a unique supplier. The distributed database network configuration is shown in the Figure 3. Note that the relational tables could be replicated at more than one node, depending on the database design. The following query can be answered from the database.

"Print the names and cities of the suppliers, that supply over 400 screws to the projects in Athens, alongwith the weights and actual quantity of screws supplied."

As mentioned earlier, the query can be fulfilled in various ways. Let us say that a particular query optimization algorithm used on this query generated the query processing algorithm described in Figure 4. If the database access primitives and communication routines are, as explained in Figure 5, then the



Project (S.Sname,S.Scity,P.Weight,SPJ.Qty) Node=2,file=12
Join (P.P# = SPJ.P#) Node=2,file=11
Project (P.P#, P.Weight) Node=2,File=2
Project (SPJ.P#,S.SName,S.Scity,SPJ,Wt) Node=3, File=10
Select (P.Pname=Screw) Node=2, FIle=1
Semijoin (J.J# = SPJ.J#) Node=3,file=9
Project (J.J#) Node=1,file=4
Project (SPJ.J#,SPJ.P#,S.Sname,S.Scity,SPJ.Qty) Node=3,file=7
Select (J.Jcity=Athens) Node=1,file=3
Join (S.S# = SPJ.S#) Node=3,file=6
Project (S.S#,S.Sname,S.Scity) Node=3,file=5
Select (SPJ.Qty>400) Node=4,file=6
P (P#,Pname,Wt,Pcity) Node=2
J (J#,Jname,Jcity) Node=1
S (S#,Sname,Status,Scity) Node=3
SPJ (S#,P#,J#,Qty) Node=4

Figure 4: An Intermediate Query Tree Representaion

object code generated by traversing the tree in postorder is shown in Figure 6. Note that there are four different programs for the four host computer nodes participating in the query. Also note that the communication routines are asynchronous in nature, and can be executed parallel. i.e. if the host computer allows multitasking, and the 'sendfile' communication routine is invoked then the following task in the program can be invoked without waiting for the sendfile operation to be complete. Similarly, at the receiving site, the 'receivefile' communication routine does not perform any operation if the file is received even before the program reaches to the current control point. Otherwise, the 'receivefile' primitive makes the host program wait until the file is received. Thus, the 'receivefile' primitive acts more like a logical switch in case of multitasking.

```
Data Manipulation Language Instructions

Select  <input file name> <output file name> (select criteria)
Project <input file name> <output file name> (file attributes)
Join <input file name1> <input file name2> <output file name>(join criteria)
Semijoin <input file name1> <input file name2> <output file name>


File Transfer Instructions

Sendfile    <file name> <receiving node>
Receivefile <file name> <sending node>
```

Figure 5. Instruction Repertoire

```
Query Initiation Phase

Host 2:
    Sendfile Program 1, Host 1
    Sendfile Program 3, Host 3
    Sendfile Program 4, Host 4


Query Execution Phase

Host 1:
    Select J, File 3 (J.Jcity = Athens)
    Project File 3, File 4  (J.J#)
    Sendfile File 4, Host 3

Host 2:
    Select  P, File 1 (P.Pname = Screw)
    Project File 1, File 2 (P.Weight, P.P#)
    Receivefile File 10, Host 3
    Join    File 1, File 10, File 11 (P.P# = SPJ.P#)
    Project File 11, File 12 (S.Sname, S.Scity, P.Weight, SPJ.Qty)

Host 3:
    Project S, File 5 (S.Sname, S.Scity, S.S#)
    Receivefile File 6, Host 4
    Join    File 5, File 6, File 7 (S.S# = SPJ.S#)
    Project File 7, File 8 (S.Sname, S.Scity, SPJ.P#, SPJ.J#, SPJ.Qty)
    Receivefile File 4, Host 1
    Semijoin File 4, File 8, File 9 (J.J# = SPJ.J#)
    Project  File 9, File 10 (S.Sname, S.Scity, SPJ.P#, SPJ.Qty)
    Sendfile File 10, Host 2

Host 4:
    Select  SPJ, File6  (SPJ.Qty > 400)
    Sendfile File 6, Host 3
```

Figure 6: Query Object Code Representation

3.2 DDQS Implementation For The Sample Query

Syntax directed code translations are done as shown in Figure 7, at respective nodes in the tree traversal for the generation of application layer simulation model.

```
Receivefile :
{
  Gen-Label(sending node, file number,
          receiving node)
  Gen-Block('ASSEMBLE 2')
}

Sendfile    :
{ Gen-Block('SPLIT 2,*+2')
  Gen-Block('TRANSFER,')
  Gen-Label(sending node, file
        number, sending node)
  Gen-Macro('SEND', sending node,
            filesize, receiving node)
  Gen-Block('TRANSFER,')
  Gen-Label(sending node, file number,
        receiving node)
  Gen-Label(sending node, file number,
        sending node)
  Gen-Block('TRANSFER,*+1')
}

Join :
{ Gen-Macro('JOIN',
            processing node,
            number of input file 1 tuples,
            input file 1 tuple size,
            number of input file 2 tuples,
            input file 2 tuple size,
            number of output file tuples
            output file tuple size)
}
```

where Gen procedures emit the relevent output source code.

Figure 7: Syntax Directed Translation Code

The GPSS source code generated for the application layer of the sample query is shown in Figure 8. The code translations for DML operations are very similar to the query processing algorithm in Figure 6, except for the macro input parameters.

GPSS SPLIT and ASSEMBLE blocks are used to simulate the parallel operations of the subtasks. For example, a transaction simulating the program at Host Node 1 is split into two transactions. One of the two siblings simulate the file transfer for File 4, and then TRANSFERs to the ASSEMBLE block statement at lable 'L010403'. By forcing the transaction to transfer to the ASSEMBLE block, the synchronization is achieved between Host Node 1 and Node 3 programs for the File 4 transfer. The SPLIT-ASSEMBLE mechanism works because the transactions assembled are from the same assembly set. In fact, since many assembly sets can be active at the same time by an ASSEMBLE block, more than one database query transactions of the same query class can be activated at the same time. If the host processor allows multitasking, the other sibling transaction can start processing, while the file transfer is still in progress. Note that the application layer of DDQS is independent of the host processor architecture.

. Unique block labels must be generated along with the usage of TRANSFER block statements. In most cases, labels are avoided, if the relative block displacement is known. In other cases, information allowing the generation of unique label is used. For example, in the TRANSFER following the simulation of File 7 transfer from Node 5 to Node 3 is, a label 'L050703' is generated. There is a possibility that more than one unique labels map on to the same GPSS block. A dummy TRANFER,*+1 block is generated for all labelled statements. The DDQS also handles the simulation of the query initiation phase. For the current example three initial file transfers are made from the query initialization node to the query processing nodes. Only when the query processing node receives its program from the query initiation node, it starts the execution.

The last point to be made in the code generation is the final ASSEMBLE block in the GPSS source code for the query initiation node program (labelled as QEND, for query end). At the end of the program at each query processing node, except the query initialization node, a TRANSFER is made to the QEND ASSEMBLE block, though there exists no corresponding file transfer. The only advantage here is the automatic verification check. The QEND ASSEMBLE block ensures that all the GPSS transactions generated during the query execution process are destroyed before the final transaction is terminated. Even though the top DDQS modules are valid, the lower level simulation library routines are error prone and the consistency check may be useful for the diagnosis.

3.3 Simulation Of DML And File Transfer Processes

As mentioned before, the DML and file transfer processes are the two main constituents of the query processing algorithm, and the simulation of these processes is equally important. Fortunately, a considerable number of simulators have been written for both the processes. The approach here is brief, and only issues pertaining to the overall DDQS model are addressed.

GPSS allows the macro facility, but not the subroutine calls. To make an efficient use of it, instructions to assign the input macro parameters to the transaction parameters are expanded first. The

```
     GENERATE ,,,1,5000                    *  QUERY TRANSACTION GENERATION
***** QUERY INITIATION PHASE **************
     SPLIT 1,*+2                           *
     TRANSFER,L021302                      *
 SEND MACRO 2,5,1                          *   SEND PROG 1 TO NODE 01
     TRANSFER,L021301                      *   JUMP TO LABEL L021301
 L021302 TRANSFER,*+1                      *
     SPLIT 1,*+2                           *
     TRANSFER,L021402                      *
 SEND MACRO 2,5,3                          *   SEND PROG 3 TO NODE 03
     TRANSFER,L021403                      *   JUMP TO LABEL L021403
 L021402 TRANSFER,*+1                      *
     SPLIT 1,*+2                           *
     TRANSFER,L021502                      *
 SEND MACRO 2,5,4                          *   SEND PROG 4 TO NODE 04
     TRANSFER,L021504                      *   JUMP TO LABEL L021504
                                           *
***** QUERY EXECUTION PHASE ***************
 L021504 TRANSFER,*+1                      *
 SELECT MACRO 2,1000,50,80                 *   P.PNAME = SCREW
 PROJECT MACRO 2,50,80,8,80                *   (P.WT, P.P#)
 L031002 ASSEMBLE 2                        *   RECEIVE FILE 10 FROM NODE3
JOIN MACRO 2,8,80,40,2000,44,16000        *  P.P# = SPJ.P#
 PROJECT MACRO 2,44,16000,40,16000         *   (SNAME, SCITY, WEIGHT, QTY)
 QEND ASSEMBLE 4                           *   END OF QUERY PROCESSING
     TERMINATE 1                           *
                                           *
******* PROG AT NODE 01 *******************
 L021301 TRANSFER,*+1                      *   QUERY BEGINS AT NODE 01
 SELECT MACRO 1,46,100,46,20               *   J.JCITY = ATHENS
 PROJECT MACRO 1,46,20,4,20                *   (J.J#)
     SPLIT 1,*+2                           *
     TRANSFER,L010401                      *
 SEND MACRO 1,10,3                         *   SEND FILE 04 TO NODE 03
     TRANSFER,L010403                      *   JUMP TO LABEL L010403
 L010401 TRANSFER,*+1                      *
     TRANSFER,QEND                         *   END OF PROG AT NODE 01
                                           *
******* PROG AT NODE 03 *******************
 L021401 TRANSFER,*+1                      *   QUERY BEGINS AT NODE 03
 PROJECT MACRO 3,50,50,46,50               *   (SNAME, SCITY, S#)
 L040603 ASSEMBLE 2                        *   RECEIVE FILE 06 FROM 04
 JOIN MACRO 3,46,50,16,1000,58,10000       *   S.S# = SPJ.S#
 PROJECT MACRO 3,58,10000,54,10000         *   (SNAME, SCITY, P#, J#, QTY)
 L010403 ASSEMBLE 2                        *   RECEIVE FILE 04 FROM 01
 SEMIJOIN MACRO 3,54,10000,4,20,54,20000   *   J.J# = SPJ.J#
 PROJECT MACRO 3,54,20000,40,20000         *   (SNAME, SCITY, SPJ.P#, QTY)
     SPLIT 1,*+2                           *
     TRANSFER,L031003                      *
 SEND MACRO 3,40,2                         *   SEND FILE 10 TO NODE 02
     TRANSFER,L031002                      *   JUMP TO LABEL L031002
 L031003 TRANSFER,*+1                      *
     TRANSFER,QEND                         *   END OF PROG AT NODE 03
                                           *
******* PROG AT NODE 04 *******************
 L021504 TRANSFER,*+1                      *   QUERY BEGINS AT NODE 04
 SELECT MACRO 4,16,2200,16,1000            *   SPJ.QTY > 400
     SPLIT 1,*+2                           *
     TRANSFER,L040604                      *
 SEND MACRO 4,100,3                        *   SEND FILE 06 TO NODE 03
     TRANSFER,L040603                      *
 L040604 TRANSFER,*+1                      *
     TRANSFER,QEND                         *   END OF PROG AT NODE 04
 ******************************************
```

Figure 8: GPSS Program Source Code For The Applicaion Layer Of Sample Query

actual DML or file transfer simulation process is then invoked. The call interface is handled by storing the return block address in one of the transaction parameters.

3.3.1 DML process Model. A relational DML operation is a classical problem of computing system simulation. One or more files are accessed using file management system of the operating system or the database management system, and an output file is written on the disk. The number of accesses are the function of the application dependent characteristics, such as the number of input and ouput tuples or attributes as well as the system dependent characteristics, such as the file structers used or the database access methods. Three different access methods are common in the database management systems.

i. nested loop
ii. sort/merge
iii. hashing

In DDQS, the host processor, memory and channels are modelled. The model cycles a DML transaction through the phases read-input, process and write-output blocks.

3.3.2 File Transfer Model. File transfer protocols are dealt with in the application layer of the ISO/OSI communication model as described in Tanenbaum(1981). The simulation model for the file transfer can be detail by explicitly simulating many of the ISO/OSI protocol layers. Simulation models also differ for different architectures. Library simulation routines can be written for local area networks, point to point networks or broadcast networks.

Many of the communication protocol layer algorithms are event driven. e.g. in a typical network layer protocol, a communication processor is invoked by the host if there is any data to be sent, or by a remote communication processor attached by a link. Once invoked, the communication processor performs functions depending on the type of invokation and the network configuration status. The processor may or may not disable itself for further invokations until the current operation is complete.

The conventional GPSS world view is transaction driven and may not be ideal for the simulation of protocol layer algorithms. Instead a modified version of the process driven view discussed in Henriksen (1982) is employeed. In this method, a process is represented by a GPSS transaction in a continuous loop. The transaction is controlled with the help of GPSS storages. Note that the transactions controlling the behaviour of the process transaction may themselves be either process transactions or transactions external to the model. Thus a hybrid version of both transaction and process views is possible. For example, the application (top) layer of the DDQS uses conventional GPSS world view in mapping GPSS transactions to the database query transaction. The inner layers use process view by mapping the protocol processes to the GPSS transactions. Additional benefits are the reduction of number of active GPSS transactions and subsequent process cost.

4. FUTURE WORK

The current DDQS is used to simulated only one class of queries against a background workload. Interesting results are anticipated from the simulation of more than one classes of queries. These experiments are typically of interest to guide the research effort on intelligent query processing algorithms based on the interaction of different classes of queries. Typically, a knowledge base could be built from the learned experience.

The current version of DDQS simulates only the nested loop DML access methods and point to point message switching network architecture. The DDQS needs to be enhanced by incorporating simulation of different database structures and network architectures in the DDQS simulation library.

Note that any good database management system tries to evaluate as many DML operations as possible in one datafile access. For example, the Select and Project on the JOB file at Node 1 are combined together. Even further, some effeciency could be achieved by receiving the communication files in the main memory rather than on the disk, thus reducing a number io operations to store temporary files. However, this requires a distributed operating system that merges the communication functions with the on site processing functions.

The DDQS models only retrievals from the distributed databases. In some databases updates and retrievals are performed concurrently. This complicates the issue further, since the database integrity has to be maintained. DDQS could then be a part of the simulator encompassing all areas of the distributed database systems.

ACKNOWLEDGEMENTS

I am very grateful to Dr. C.S. Egyhazy, Virginia Tech, Northern Virginia Graduate Center, Mr. Jim Henriksen, Wolverine Software Corporation, and Mr. Roy Wnek, Booz-Allen and Hamilton, for their invaluable suggestions in the design and development of the model.

BIBLIOGRAPHY

Date, C. J. (1986). Introduction To Database Management System, Vol I and II. Addison Wesley.

Henriksen, James O. (1981). GPSS - Finding The Appropriate World-View. Winter Simulation Conference.

Henriksen, James O. and Crain, R. C. (1983). GPSS/H User's Manual. Wolverine Software Corporation. Annandale, Virginia.

Tanenbaum, A. (1981). Computer Networks. Prentice Hall.

Aho, and Ullman, J. D. (1977). Principles Of Compiler Design. Addison Wesley.

Wnek, R. M. and Roth, P. F. (1982). Simulation Of A Distibuted Database System Incorporating A Routing Optimizer. Winter Simulation Symposium.

Wnek, R. M. (1984). Extension and Application of a Distributed Simulator. Virgina Tech, Northern Virginia Graduate Center (Unpublished)

AUTHOR'S BIOGRAPHY

RAJU KOCHAREKAR is a Software Consultant in the IBM Facility Center at The World Bank. He received a B.Tech. in Electrical Engineering from Indian Institute of Technology, Bombay in 1981, and an M.S. in Computer Science from Virginia Tech, Virginia in 1986. From 1981 to 1985, he was in Tata Burroughs Ltd, Bombay as a Software Engineer.

Raju Kocharekar
The World Bank
1818, H street NW,
Washington, DC 20433
(202) 473-2153