A Resource Broker for Optimal Site and Query Scheduling in a Distributed Relational Database System

Vijayakumar B.
Computer Science and
Engineering Department
Regional Engineering College
Tiruchirappalli, India
vijay@rect.ernet.in

Gopalan N.P.
Computer Science and
Engineering Department
Regional Engineering College
Tiruchirappalli, India
gopalan@rect.ernet.in

ABSTRACT

In this paper, a **Resource Broker** is proposed over a *distributed query processor* for handling queries from different client sites in an efficient manner. It is distributed in operation and takes optimal decisions with regard to **site allocation** and **query scheduling**. When a query is submitted for execution, it is first checked for correct syntax and then broken down into **operation_schedules** or **atomic operations**. The **Resource Broker** proceeds to construct the **operation_schedule tree** which specifies the sequence in which the *atomic operations* are performed. It also balances the *load* at the different sites by interaction with it's **peer resource brokers** at these places and by using *dynamic system parameters.* The system has been tested successfully with multiple servers and clients and it's performance has been analyzed.

1. INTRODUCTION

In a distributed query processing environment, it is necessary to devise strategies for exploiting **concurrency** while executing multiple queries which are initiated from the **client** sites. The host systems in the network should use appropriate **admission** and **allocation** policies for handling the queries efficiently. It is also essential to decide the sites at which various **atomic operations** such as *select, project, independent predicate execution*, etc. are to be performed. In this paper, a **Resource Broker** is proposed over a **Distributed Query Processor** (RBDQP) for efficient handling of queries from different **client** sites. This module acts as a manager of resources for competing operations in a distributed environment and takes optimal decisions with regard to **site allocation** and **query scheduling**.

2. RELATED WORK

The concept of **query scheduling** derives it's origin from *operating system scheduling* where the emphasis is on measuring an algorithm's efficiency based on it's average *turnaround, response* and *wait times.* The idea of *intra* and *inter query concurrency* is discussed in [3], [10], [11]. Martin, Lam and Russel [8] suggested four different strategies for **site allocation** viz., *Branch and Bound, Simulated Annealing, Local Search* and *Greedy.* The use of **global query completion times** and **query initiation delays** as performance metrics in this context are discussed in [6]. The use of a **Resource Broker** for a centralized database environment with *multiple users* and *mixed workloads* is dealth in greater detail in [1], [5]. The present work generalises this concept for a **distributed environment**.

3. DEFINITIONS

3.1 Independent Predicates

Independent Predicates are predicates which have conditions defined over a single relation [12]. For example, $\mathbf{R}_i.\mathbf{A}_j$ **<= a constant** is an **independent predicate** whereas, $\mathbf{R}_i.\mathbf{A}_j <= \mathbf{R}_k.\mathbf{A}_i$ is not **independent** because this involves more than one relation. Here $\mathbf{R}_i$ and $\mathbf{R}_k$ are relations, $\mathbf{A}_i$ and $\mathbf{A}_j$ are attributes and **i, j** and **k** are integers.

3.2 Chain Query

A **Chain query** is a special type of query for which the general acyclic hypergraph is a chain; (i.e.) for that hypergraph, it's edges can be listed in an order $\mathbf{R}_1, \mathbf{R}_2..., \mathbf{R}_n$ such that the only non-empty intersections are between $\mathbf{R}_i$ and $\mathbf{R}_{i+1}$ for $1<=i<n$. Also, the relation to be reduced is at one end, say R_1 [3], [11].

ADVANCES IN DATA MANAGEMENT 2000
Krithi Ramamritham, T.M. Vijayaraman (Editors)
Tata McGraw-Hill Publishing Company Ltd.

3.3 Operation_Schedule

An **Operation_Schedule** represents the smallest schedulable unit in a query which can be executed independently. The **length** of an operation_schedule is defined as the number of steps required to compute it's result.

Table 1 shows the valid **operation_schedules** and their corresponding **lengths**:

Table 1

Operation_Schedule	Length
Select	1
Project	1
Send/Receive Fragment(Relation)	1
Merge Fragment	1
Independent Predicate Execution	1
Join	> 1

The **length** of the **Join Operation_Schedule** is greater than 1 since it involves a sequence of steps to constitute the *candidate relations* before performing the **join** operation.

3.4 Multi-Programming Level

In the context of a DBMS, **Multi-Programming Level** (**MPL**) is defined as the number of *light-weight co-operating processes* which are involved in **operation_schedule execution**, executing concurrently in the System [5]. As and when an **operation_schedule** *enters(leaves)* the system, the MPL value is *incremented(decremented)* according to it's *length.*

3.5 MPL_Threshold

MPL_Threshold specifies a ceiling on the **MPL** value of a system. Scheduling of requests cannot exceed this limit, so as to avoid deadlocks for resources among any number of competing queries.

3.6 Safe_Limit

Safe_Limit corresponds to a particular **MPL** value beyond which the system reroutes *client* queries. **Operation_Schedule requests** will be accepted but not *full queries.*

3.7 Distributed Multi-Programming Level

In a distributed database system with M sites, **Distributed Multi-Programming Level** (**DMPL**) is defined as the sum of **MPL** values at each site **i**, where $\mathbf{i} = \mathbf{1}, \mathbf{2}, \ldots, \mathbf{M}$. $\mathbf{DMPL} = \sum_{i=1}^{M} \mathrm{MPL}_i$, at any instant when the measurement is carried out. The **DMPL** value is stored at each site and is updated by the resource broker as and when the **MPL** value changes at any given site.

DMPL_Threshold specifies the maximum limit on the value of **DMPL** in the overall distributed database system.

3.8 Queue_Length

For any given site **i** ($\mathbf{i} = \mathbf{1}, \mathbf{2}, \ldots, \mathbf{M}$), the **Queue_Length** defines the number of **queries / operation_schedules** waiting in a queue for subsequent execution. It is *incremented (decremented)* by 1 when a **query / operation_schedule** *enters (leaves)* the queue.

3.9 Initiation Delay

Initiation Delay [6] is defined as the time spent by a **query / operation_schedule** in the **queue** before it is scheduled for execution.

3.10 Global Query Completion Time

The **Global Query Completion Time** [6] refers to the time interval between the *arrival of the first query* and the *completion of the last query* measured across the distributed system.

4. ASSUMPTIONS

There are **M** sites and the **global schema** maintains information on *identification of the sites, the names of the relations* and their *attributes* and the *predicates* over which the *horizontal fragments* are created. Each site contains the same copy of **global schema**. The queries are assumed to be in standard SQL format.

5. RBDQP SYSTEM ARCHITECTURE

The proposed system **RBDQP** is intended to meet the following requirements:

- Design efficient policies for **admission** and **allocation** in query scheduling.
- Design a suitable *site allocation* scheme incorporating *parallelism* in the *query execution plan* and *load balancing* at the various sites.

The various components of the **RBDQP** System (cf. Figure 1) are explained below:

5.1 Distributed System Data Manager (DSDM)

The **DSDM** exchanges the various *distributed database parameters* between the sites periodically and finds the **optimal order** of the **join** predicates. The parameters considered are:

Statistical Data : *cardinality of the relations* and *domain range of the join attributes.*

Dynamic Data : *creation of new indexes*, *loads at different sites* and a *global image* of the available memory resources (Primary and secondary memories).

5.2 Resource Broker

A **Resource Broker** acts as a manager of resources for competing operations in a distributed environment. When a query is submitted for execution, it is first checked for correct syntax and then broken down into *Operation_Schedules (atomic operations)* like **Independent Predicate Execution, Merge Fragment, Join** and so on. The **Resource Broker** proceeds to construct an **Opeartion_Schedule tree** which specifies the sequence in

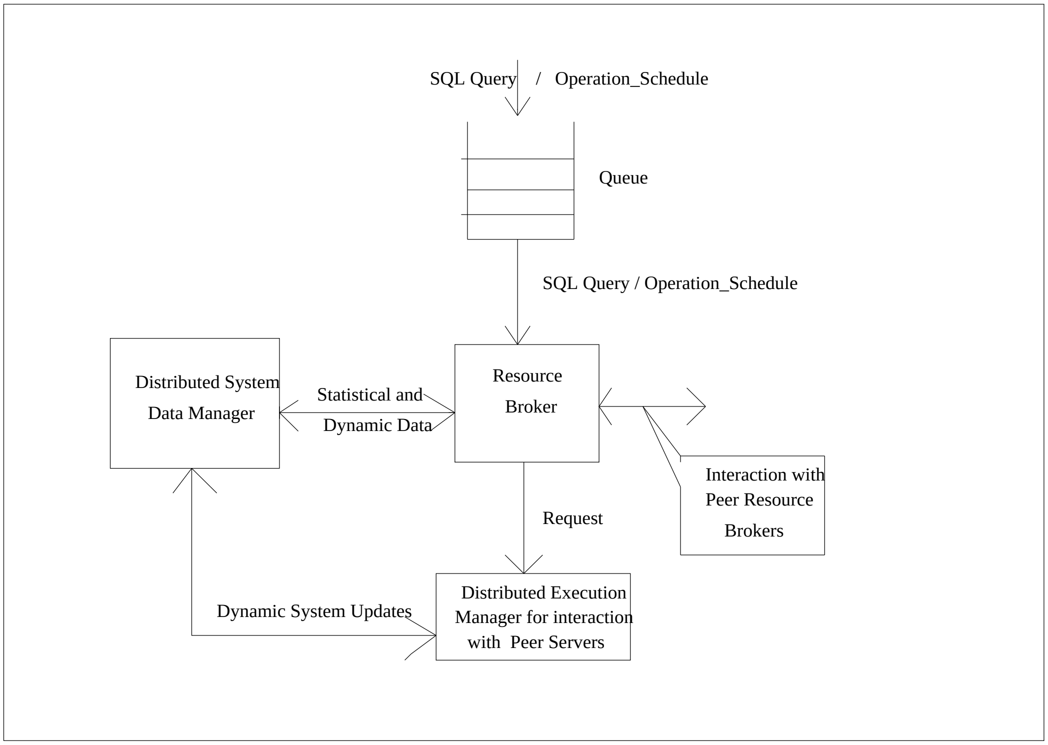

Figure 1: RBDQP System Architecture

which the *atomic operations* are performed. Instead of executing all the *atomic operations* in the same server which will naturally increase it's load, they are distributed to other servers for execution based on load details provided by the **dynamic resource broker** residing at every server. The **resource broker** always tries to make an efficient utilization of resources available at the various sites. It also interacts with *peer resource brokers* present at these sites and exchanges their **MPL** values.

5.3 Distributed Execution Manager (DEM)

The **DEM** receives the following types of requests :

1. Requests from the **clients** for query execution.
2. *Operation_Schedule* Requests from another **server** for partial execution of the query, such as

- **Independent Predicate** Execution.
- **Shipping** a fragment of a *relation*.
- **Merging** the fragments of a *relation*.
- **Join** Operation at the local site or at a remote site.

It issues commands to perform elementary operations such as *select, project, merge* and *join*.

6. ALGORITHM : RBDQP

The following algorithm is for **site i** and is executed when a client makes a **SQL query** request or a peer server makes an **Operation_Schedule** request:

Input : **SQL query** from a **client** site or an **Operation_Schedule** Request from a peer server,
Maximum Queue_Length, **MPL_Threshold**.
/* **Operation_Schedule** corresponds to any one of the operations mentioned in Table 1 */
/* For the experimental setup, **Maximum Queue_Length** = 20 and **MPL_Threshold** = 20 */

Output : Result of the *Full Query / Operation_Schedule.*
Procedure

```
begin
1. Accept a request.
2. If the request is for SQL query execution then
      begin
       If ( Queue_Length <= ( Maximum Queue_Length - Safe_Limit) ) then
                    Call Add_To_Queue(Request)
          else
                  Reroute this request to another server with lesser load.
                       Reject it when rerouting is not possible.
          endif
      end
   else
         /* Operation_Schedule request from a peer server */
      begin
       If ( Queue_Length <= ( Maximum Queue_Length
```

```
) )               then
                      Call Add_To_Queue(Request)
             else
                      Hold the request until there is a free slot in the queue.
                      Call Add_To_Queue(Request)
             endif
         end
   3. Call the Resource_Broker.
   4. Transfer control to the Distributed Execution Manager.
   end.
```

In order to implement *query scheduling*, the **Resoure Broker** uses two policies namely, **Admission** and **Allocation**. The **Admission Policy** decides the number of *queries / operation_schedules* that can operate concurrently in the *distributed database system.* For any given query, the **Allocation Policy** decides the *Processor* and allocates *Memory.* A *query / Operation_Schedule* is forced to the head of the queue irrespective of the policy, if it's waiting time exceeds that of the others. This is done to take care of the *indefinite wait* problem.

6.1 Add_To_Queue(Request)

```
begin
      Increment Queue_Length by 1.

         If the request is an Operation_Schedule then

            Increment MPL value by length of Operation_Schedule

         Endif

      If the admission policy is FIFO then
         Add the request at the rear end of the queue
      else
         Place the request at the first available empty location in the queue
      endif
end.
```

6.2 Resource_Broker

```
begin
1. Check the queue status.
2. If the queue is empty or the
   (DMPL value > DMPL_Threshold )
   then sleep for a finite amount of time and go to step 1.
3. If the admission policy is FIFO then Remove a request which is at the front end of the queue.
4. If the admission policy is Priority then evaluate the priorities of all the entries in the queue. Remove the request having the highest priority from the queue. If a class of requests has the same priority, then FIFO is used within the same class.
5. If the admission policy is Heuristic then apply the Heuristic_Function on all the entries present in the queue. Select a request that has an optimal heuristic.
6. Call Allocate_Memory for this request.
end.
```

6.2.1 Heuristic_Function

An **Operation_Schedule** is executed in the increasing order of it's **length** and is having a *higher priority* over a *full query.* The **Operation_Schedule** requests are to be executed immediately to avoid deadlock and indefinite wait problems.

For a **full query**, the following heuristics are used:

1. A query having large number of **independent predicates** is given the topmost priority since it reduces the size of the intermediate *relations.* It minimizes the *communication cost* and also improves the *response time* since these predicates can be executed in **parallel** at the different sites.

2.A **chain query** with a lesser number of **joins** can be given high priority since this would result in minimum *processing cost.*

Using the above *heuristics* and the *static cardinalities* of the fragments of the input *relations* obtained from the **global schema**, the queries are quantified in the increasing order of the **processing cost**.

6.2.2 Allocate_Memory

The **Distributed System Data Manager** maintains a global image of the available *memory* resources (Primary and secondary memories). A maximum of 16 MB memory is assumed to be available at each site.
The **Global_Maximum_Memory** is computed as follows : $\sum_{i=1}^{M}$ Local_Maximum_Memory*(i)*, where M represents the number of sites. An **Operation_Schedule** is allotted a finite amount of memory equal to :
$(Local\_Maximum\_Memory)/(2*MPL\_Threshold - SafeLimit)$ ...... (1)
It is to be noted that in equation (1), the denominator uses *(MPL_Threshold + (MPL_Threshold - Safe Limit)).* This is done to have at hold some memory which could be allotted to the immediate *operation_schedules.* Hence, the fairness is towards **atomic operations**. When the memory requirement for an operation exceeds the allotted value, it sends a request to the **resource broker** which allots *memory* if available. Otherwise, the operation is blocked. The *waiting time* is also included in the **global query completion time**.

7. COMPLEXITY

The following parameters are considered in the analysis of the **RBDQP algorithm :**
N : number of relations in the query
M : number of sites
n : number of tuples in a relation/fragment
sf_j : selectivity factor of the j^{th} relation, for the selection operation = number of tuples selected/n
c : cost of transmitting a relation/fragment from one site to another
n_i : number of tuples joined at stage i, in the reduction of the chain
$merge$: time for merging the fragments of a relation $\propto \mid n \mid$.

The time complexity of the algorithm is given by :
$O(s) + N * ((M-1)O(c) + O(merge)) + (N-1)*O(c) + \sum_{i=1}^{N-1} O(n_i log n_i)$ .... (2)
where $O(s) = Max\ [\ O(sf_j * n)\ ]$, $1 <= j <= N$.

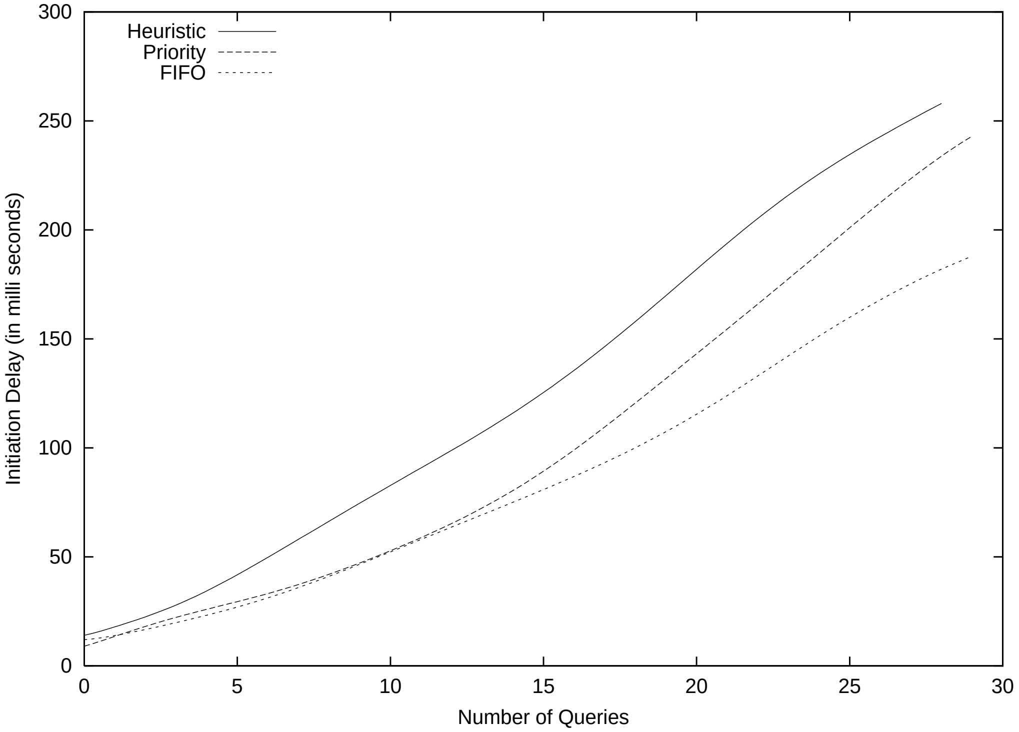

Figure 2: Resource Broker Overhead(Maximum Queue_Length = 20, MPL_Threshold = 20)

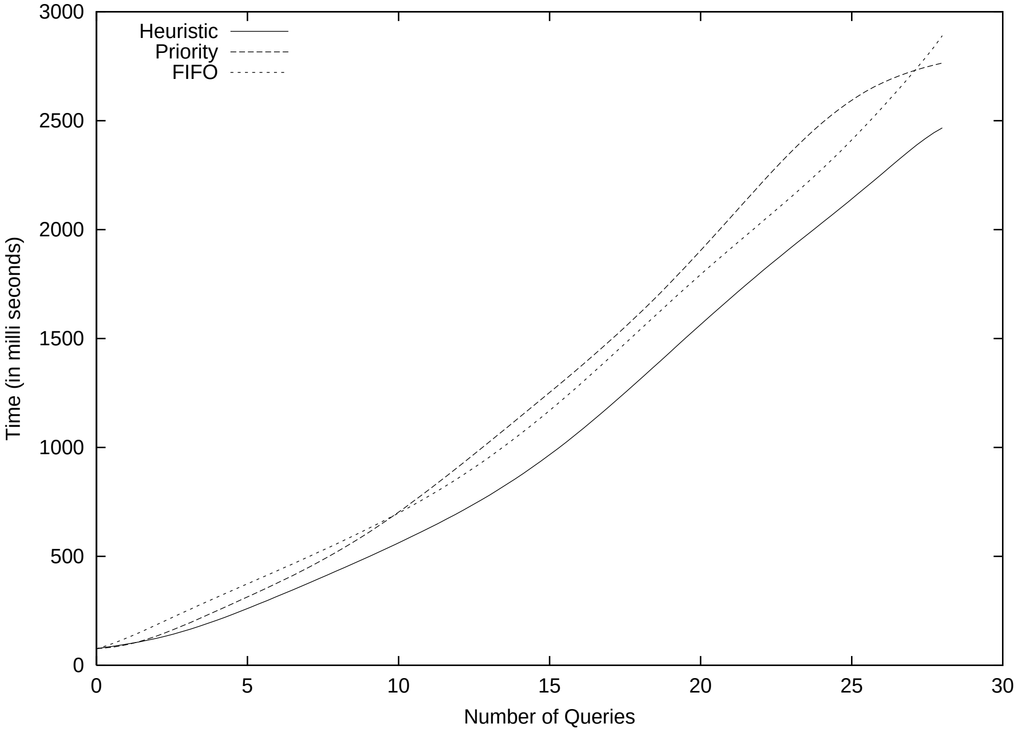

Figure 3: Global Query Completion Times(Maximum Queue_Length = 20, MPL_Threshold = 20)

The first term in equation (2) corresponds to cost of executing the **independent predicates**; the second term corresponds to the cost of **merging** the fragments of the **N** input *relations* stored at **M** sites; the third term corresponds to the **transmission cost** for **N-1** relations and the last term corresponds to the **join cost** for **N-1** relations.

8. IMPLEMENTATION

The M different sites are identified with different ***IP* addresses** and the *distributed server* runs at each site. When the *server* is started, it intializes the local buffer with *local* and *global schema* table values. The **client-server** model [7] is used in the implementation of the RBDQP System. This system has the ability to process requests concurrently from **clients** as well as from other **distributed servers**. The implementation is carried out on a network of **unix** systems. The network protocol used for communication between the various **servers** as well as between the **servers** and **clients** is TCP/IP [4], [9]. Since, TCP/IP is widely supported, this model is extendible easily to hosts spread across local and wide area networks. The algorithm is implemented using **C**.

9. TEST CONFIGURATION

The RBDQP System has been tested successfully under mixed workload conditions. The input to the system consists of thirty SQL queries on five *relations* whose fragments are spread over four *sites.* The *query mix* is uniformly distributed over *Single Select, Single Join* and *Chain Queries.* The *Chain Queries* themselves include **Select** and **Project** Operations. The database size is varied in the range 20 MB - 40 MB and the results obtained were qualitatively similar.

10. SITE ALLOCATOR

The query submitted by the client is executed in the following order :

(a) **Independent Predicates** Execution .

(b) **Merging** the fragments of the relations.

(c) Performing the **join** operations in an order as decided by the **Distributed System Data Manager**.

The operations within **(a)** and **(b)** are done in parallel at the different sites . The **site allocator** determines the best **merging site** for each relation. The parameters considered by the **site allocator** include :

- . Total number of fragments of each relation
- . Number of tuples in each fragment
- . Load Balancing Factor (LBF)
- . Performance Factor of the system .

LBF tries to distribute the **Merge Fragment** requests for all the relations uniformly across the sites, thereby increasing the extent of parallelism and minimizing the time taken to complete the merge phase. For proper load balancing, the number of relations considered for merging at a site should not be greater than $\lceil N/M \rceil$, where N is the number of relations involved in the query and M is the number of sites.

The **Performance Factor** for any site **J** is defined as

PF(J) = MPL(J) + Queue_Length(J), $1 <= J <= M$.

Consider **N** *relations* whose fragments are distributed over **M** *sites.* Each entry of the matrix **CARD_TABLE(I, J)** specifies the number of tuples of *Relation* **I** present at *Site* **J**, where (**I = 1, 2, ..., N**) and (**J = 1, 2, ..., M**).

The matrix **LOCAL(M, M)** has an entry '**0**' for all diagonal elements and the remaining elements have a value of '**1**'. '**0**' entry indicates that a fragment of a given *relation* is present at the *local site* and does not involve any *communication cost.* '**1**' entry indicates that a fragment of a given *relation* contributes for *communication cost*, as it is shipped to a *remote site*, for **merge** operstion.

The *performance* of any *site* is represented by a diagonal matrix **PF(J, J),** where (**J = 1, 2, ..., M**). The entries in the diagonal represent the *performance factors* of the various *sites* as explained earlier.

The *communication cost* for merging the *relation I* at *site J* is given by the matrix **COMM_COST(I, J)**, where (**I = 1, 2, ..., N**) and (**J = 1, 2, ..., M**).

COMM_COST(I, J) = CARD_TABLE(I, J) * LOCAL(J, J). The *overall cost* for **merging** a *relation **I*** at *site **J*** is given as follows :

MERGE_COST(I, J) = (COMM_COST(I, J)) * PF(J, J), where (**I = 1, 2, ..., N**) and (**J = 1, 2, ..., M**).

From the above matrix, the **optimal site** for merging a *relation* is obtained as follows:

MERGE_SITE(I) = MIN(MERGE_COST(I, J)), where (**I = 1, 2, ..., N**) and (**J = 1, 2, ..., M**).

The **independent predicates** are executed in the first phase. In the second phase, the **Merge Fragment** requests are carried out. In the final phase, all **join** predicates will be passed to the **DSDM** which finds their **optimal join order**.

The **optimal join order** is determined as follows:

1. The given input *relations* are ordered in the increasing order of their *sizes.*

2. For each **join** operation, estimate the size of *intermediate results*, assuming that the *join selectivity factor is 1*; (i.e. the *cardinalities* of the two relations participating in the *join* operation are multiplied). In practical systems, this factor is usually between **0** and **0.5**. Choose a *join order* in which the sizes of *intermediate results* increase gradually. The resultant relation is finally sent to the **client**.

11. DISCUSSION

The performance graphs shown in Figure 2 and Figure 3, compares the three *query scheduling* approaches. From Figure 2, it can be inferred that the **heuristic** policy has more **query initiation delay** when compared to FIFO or **Priority** policies for a *fewer number of queries.* This is due to the fact that the **heuristic function** takes some time to evaluate all the entries in the queue and select an entry. As the number of queries increase the performance of the **heuristic** policy improves. From Figure 3, it can be seen that the three policies have nearly equal **global query completion times** for a smaller number of queries. This is because the system has resources sufficient enough to satisfy all the queries. As the number of queries increase, **heuristic** policy gives better **global query completion times** when compared to **Priority** and FIFO policies.

12. CONCLUSIONS

A **Resource Broker** over a *distributed query processor* (RBDQP) for optimal **site** and **query scheduling** decisions has been proposed and it's performance has been analyzed. This model has been tested with multiple servers and clients successfully. This work can be extended further to handle **replications** in *distributed relational database*, provide **fault tolerance** for the distributed servers and incorporate *concurrency control* measures for handling *update* queries.

13. ACKNOWLEDGEMENTS

The authors would like to thank Sankaran, Ramprasdh and Veerakumar for their help in experimental setup and are grateful to the referees for their helpful comments on improving this paper.

14. APPENDIX

15. REFERENCES

[1] Brown, ”Resource Allocation and Scheduling for mixed database workloads”, TR-CS-WISC-1993.

[2] Chiu D.M., Bernstein P.A and Ho Y.C, ”Optimizing chain queries in a distributed database system”, SIAM Journal of Computing, Vol 13, No. 1, Feb. 1984, pp. 116-134.

[3] Ceri S and Pelagatti G, ”Distributed Databases : Principles and Systems”, Mc-Grawhill Book Company, 1985.

[4] Comer D.E. and Stevens D.L, ”Internetworking with TCP/IP Vol. 3., Client-Server Programming and Applications”, Prentice Hall of India, 1997.

[5] Davison D.L and Graefe G, ”Dynamic Resource Brokering for Multi-User Query Execution”, ACM SIGMOD 1995, San Jose, CA USA, pp. 281-292.

[6] Freider O and Baru C.K. , ”Site and Query Scheduling Policies in Multicomputer Database Systems”, IEEE Trans. on Knowledge and Data Engg., Aug. 1994, pp. 609-618.

[7] Khoshafian S., Chan A., Wong A. and Wong H.K.T., ”A Guide to Developing Client/Server SQL Applications”, Kaufmann Pub., 1992.

[8] Martin T.P, Lam K.H and Russel J.I, “An Evaluation of Site Selection Algorithms for Distributed Query Processing”, The Computer Journal, Vol 33, No. 1, 1990, pp. 61-69.

[9] Richard Stevens W, ”Unix Network Programming”, Prentice Hall of India, 1996.

[10] Tamer Ozsu M. and Valduriez P, ”Principles of Distributed Database Systems”, Prentice-Hall Inc., 1991.

[11] Ullman J.D., ”Principles of Database Systems”, Edition 2, Computer Science Press, INC., 1995.

[12] Vijayakumar B and Gopalan N.P., ”Distributed Query Optimization using Independent Predicates and Detachment for the Relational Model”, Proceedings of the National Conference on Advanced Databases and Applications, SCSE, Anna University, Chennai, Oct. 1998.