In distributed database, it is important to take data locality into account since they have big impact on the communication requirements. Several techniques are proposed for balancing the load in homogeneous applications but still some improvement in terms of efficiency is required. In this chapter, a load balancing architecture that can deal with homogeneous applications in distributed database more efficiently is presented. In the proposed architecture, memory and CPU utilization based priority method is used and data locality is also taken into consideration along with process waiting time and data transmission time. A load balancing algorithm has been developed to balance load on different nodes working in homogeneous environment in a fragmented distributed database.

4.1 INTRODUCTION

With the development of new technologies, the computing problems we need to solve have become more complicated and larger in size. This has motivated us to move towards deploying distributed systems to resolve our problems. The goal is attainable by partitioning problems into executable processes that can be executed on multiple nodes [SSH2008]. Load balancing is the process of improving the performance of a parallel and distributed system through a redistribution of load among the nodes [PWE2006] [SDH2007].

In general, a scheme of load balancing has three phases: information collection, decision making and data migration. In the first phase, information about load distribution and the state of computational environment is collected. In the second phase, we compute the optimal load distribution while in the migration phase, the extra desired load on a node is transferred to a node with a load below the desired level [ADS1999].

In this section, several algorithms each suitable for a different type of task which have been implemented are described. In chunk scheduling or self scheduling [CKR1985], a constant number of tasks [CDP1987] are considered; the number is proportional to number of remaining tasks in the queue. Factoring [SFH1991] takes into account both the number of tasks in the queue and the number of processors used by the application. Hua-
Chapter 4: Priority and Data Locality Based Dynamic Load Balancing in Distributed Database

Feng Deng et al [HFD2007] presented a novel algorithm for load balancing in distributed systems in which almost all jobs are organized into the standard job combinations, each of which consisted of one to four jobs. Then they are assigned to the machines according to the assignment algorithm for job combinations, which is a special integer partition algorithm [DGR2005]. One another algorithm called Self-Training Algorithm [PMO2008] for load balancing in cluster computing used load information including CPU load, memory usage and network traffic to decide the load of each node and combined this information with properties of each job, including CPU bound, memory bound and I/O bound features that are extracted from the previous runs of these jobs.

Tapering [SLU1992] selected a chunk size based on the distribution of the observed task execution time. Tapering has been implemented on a distributed memory system. In the gradient model [FCH1987], a “node load potential” field is established over the nodes and newly created tasks drift to the node with the global minimum value through the potential field. Adaptive Contracting within Neighborhood (ACWN) [WSH1989] restricted the hop length, so that tasks are assigned to nearby nodes. Adaptive load sharing is implemented using a central manager that gathered node information periodically and redistributed tasks if needed [JXU1990]. The period is inverse-proportional to the load imbalance in the system. Reinard et al. [RVH1991] proposed an important class of load balancing methods based on data decomposition. With this method, data structures are partitioned across the nodes and each node operated on the data that is assigned to it. This method made it possible to minimize remote data references by taking data dependencies into account during data repartitioning.

An algorithm based upon hybrid technique of load balancing has been proposed which uses priority policy based on memory and CPU utilization of nodes with global task queue in distributed database to improve CPU utilization as compared to above said policies of load balancing.

4.2 PROPOSED MODEL

In the proposed model, the system consists of a set of compute nodes and a global task scheduler. The task scheduler collects the status information from the nodes and assigns task based on this information. In this system, a priority based process load balancer is used. A compute node includes data migration if data is not available at that particular
node and data transmission time for data migration to remote node is less than process waiting time at local node and also provides this status to global task scheduler and application code.

![Diagram of Priority Based Load Balancing](image)

**Figure 4.1 Proposed Model of Priority Based Load Balancing**

Different nodes are already executing different tasks or workloads assigned to them randomly or dynamically depending upon the availability of the node at that instance of time. In our proposed (Neemon) algorithm, server decides about the node to which the next job is to be given. Server maintains the status of current memory utilization of all the nodes and the node having maximum memory available and minimum CPU cycle usage is chosen as the node to which the next job is to be assigned. So, depending upon memory and CPU utilization, a node with lowest CPU cycle utilization and maximum memory available at that particular instance of time is given the highest priority amongst all other nodes and a highest priority number is assigned to it.

The information regarding each client node in terms of node IP address, number of jobs running concurrently, memory used, free memory available, number of data items accessible and CPU cycles used is maintained by server as shown in Table 4.1. Each client node maintains and stores information about each data item accessible at that node in detail as illustrated by Table 4.2.
Chapter 4: Priority and Data Locality Based Dynamic Load Balancing in Distributed Database

Table 4.1 Details of Client Nodes Available at Server

<table>
<thead>
<tr>
<th>Seq. No.</th>
<th>Node ID</th>
<th>Number of Job running</th>
<th>Memory used</th>
<th>Free memory</th>
<th>Number of data items at each node</th>
<th>CPU cycles used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 Data Items stored at Client Node

<table>
<thead>
<tr>
<th>Node ID</th>
<th>Data item ID</th>
<th>Data type1</th>
<th>Data type2</th>
<th>Data type3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Priority set for each node keeps on changing depending upon memory usage, CPU utilization and jobs finished, periodically. Updates are being made in Table 4.1 at the server, so that the current status of highest priority node is readily available for new job assignment.

4.3 WORKING OF PROPOSED ALGORITHM

In the proposed algorithm, when a new request arrives at the server, it firstly checks for nodes which are having requested data available locally. Then the priority \( p[i] \) is assigned to each such node \( i \) and it is checked to get the information about the memory usage, free memory available, idle CPU cycles.

The priority level is checked against a minimum priority level called a threshold value \( \alpha \) and node with priority less than threshold value \( \alpha \) is considered overloaded. So, if the priority \( p[i] \) of selected node \( i \) is above the threshold value \( \alpha \), then new process is assigned to that node even if it is not the highest priority node available. But if priority of that node is less than the threshold value \( \alpha \), then there are two considerations:

1. Another node with higher priority i.e. the node with priority \( \alpha + 1 \) is selected for job assignment.
2. The process waiting time \( (t_w) \) at local node \( i \) and data transmission time \( (t_d) \) per hop from local node \( i \) to newly selected remote node with priority \( [\alpha + 1] \) is calculated. The newly selected node is efficient in executing the new job assigned because its priority is above the threshold value \( (\alpha) \), but now it is also a remote node where data is not available locally, so the required data need to be transmitted to node with priority \( [\alpha + 1] \) which increases the data transmission time depending upon the number of hops covered to transmit the data to destination node. NEEMON algorithm works as follow:

If \( L \) is the load to be given to the node

\[
\begin{align*}
  t_i &- \text{Processing time of node } i \\
  t_d &- \text{Data transmission time per hop} \\
  n &- \text{Number of hops}
\end{align*}
\]

Then \( T_{pt} = L(n^*t_d + t_i) \) where \( T_{pt} \) is total processing time taken by a node

**At Local Node**

\( t_d = 0 \) because data is available locally at that node.

Now, \( T_{pt} = L(n^*t_d + t_i + t_w) \) wherever waiting time \( (t_w) \) is concerned, there are two considerations:

i. In case of high priority local node, waiting time is 0 because CPU cycles and enough memory are available for new job execution.

\[
\begin{align*}
  T_{pt} &= L(0 + t_i + t_w) \\
  &= L(0 + t_i + 0) \\
  &= L(t_i)
\end{align*}
\]

ii. If the local node is low priority node, then waiting time is a major factor in total processing time of a node.

\[
\begin{align*}
  T_{pt} &= L(0 + t_i + t_w) \\
  &= L(t_i + t_w)
\end{align*}
\]

Now even if data is available locally but the node is already over loaded, it may take more time to process that new job than if the same job has been executed at a remote node. Since this new job may have to wait for long in the waiting queue of the local node. So, a threshold value \( \beta \) is decided based on the fact that for how long a new job should remain in waiting queue at a local node.
Chapter 4: Priority and Data Locality Based Dynamic Load Balancing in Distributed Database

If \( t_w > \beta \) then

If \( t_w > n_t_d \) where \( n \) is number of hops between overloaded node and remote node selected for execution.

- Data migrated to remote node
- Else
  - Job is kept in waiting queue at local node
- Else
  - Job is kept in waiting queue at local node

Here two conditions are verified. First, waiting time at local node is checked against a threshold value \( \beta \) so that a process does not block for longer and if the waiting time for process at local node is greater than threshold value \( \beta \), then waiting time is checked against data transmission time to remote node. If the waiting time is less then data transmission time, the process is still made to wait in waiting queue of local node but if the data transmission time is less than waiting time, then job is migrated from local node to the higher priority node selected for execution.

At Remote Node

- \( t_d \) – Data transmission time for one hop
- \( t_i \) – Processing time of migrated job
- \( t_w \) – Waiting time of migrated job
- \( n \) – Number of hops

\[
T_{pt} = L(n \cdot t_d + t_i + t_w)
\]

At remote node, \( t_w = 0 \),

So, the total processing time of the migrated job is calculated as:

\[
T_{pt} = L(nt_d + t_i + 0)
\]

Neemon Algorithm

1. For new request
   a. Check for node \( i \) having required data.
   b. Check the priority \( p[i] \) of node \( i \) having required data.
   c. Set threshold value \( \alpha \) for priority level.
   d. Set threshold value \( \beta \) for waiting time.
   e. At local node \( i \),

55
Chapter 4: Priority and Data Locality Based Dynamic Load Balancing in Distributed Database

Priority level is checked

\[ p[i] > \alpha \]

New job is assigned to node \( i \).

Else

{\}

Node with priority \( \alpha + 1 \) is selected

If \( t_w > \beta \) then

\[ t_w > nt_r \]

Data migrated to remote node with priority \( \alpha + 1 \)

Else

Job is kept in waiting queue at local node

Else

Job is kept in waiting queue at local node

} } 

4.4 PERFORMANCE EVALUATION

The performance of proposed Neemon algorithm is evaluated using two measures:

1) Memory utilization w.r.t. number of processes before load balancing and after load balancing.

2) CPU utilization w.r.t. number of processes before load balancing and after load balancing.

Figure 4.2 Comparison of Memory Utilization
Chapter 4: Priority and Data Locality Based Dynamic Load Balancing in Distributed Database

Figure 4.2 shows memory utilization whereas Figure 4.3 shows CPU utilization with respect to number of processes. In Figure 4.2, at the beginning, when load balancing was yet to be implemented, different nodes were running different processes. So, memory available at each node was not known in advance. But as the number of processes increased and they were allocated to different nodes based on load balancing criteria specified, memory utilization improved gradually by almost 3-4% whereas when CPU utilization was evaluated, it increased gradually by almost 11% as shown in the Figure 4.3.

![Figure 4.3 Comparison of CPU Utilization](image)

4.5 CONCLUSION AND FUTURE WORK

A new load balancing architecture is proposed that takes into account data locality and priority for performing load balancing in an environment having applications with homogeneous tasks. NEEMON algorithm works for a fragmented database where data migration takes place when the memory and CPU cycles available at local node is not enough for process execution or the waiting time for the process is more than the execution time at a remote node. This algorithm works better in terms of memory usage and CPU utilization. Further work is needed in several areas. First, the proposed architecture has to be evaluated using more complex applications and larger systems. Second, the current algorithm works for homogeneous systems using client/server
architecture. It can be made to work for peer-to-peer networking in heterogeneous systems also.