Data Allocation in Distributed Database Systems

Technical Report No.: DBRG.RB-ST.A50715
Issue Date: Summer 2005

Reza Basseda
Samira Tasharofi

Abstract
The aim of a data allocation algorithm is to fix the sites where the fragments are located so as to minimize the total data transfer cost, under the storage constraints (i.e., the maximum number of fragments that can be allocated at a site) at each of the sites. In this report we explore the distributed database allocation problem, which is intractable. We will also describe and analyze different data allocation algorithms proposed till now.

1. Introduction

Developments in database and networking technologies in the past few decades led to advances in distributed database systems. A DDS is a collection of sites connected by a communication network, in which each site is a database system in its own right, but the sites have agreed to work together, so that a user at any site can access data anywhere in the network exactly as if the data were all stored at the user's own site.
The primary concern of a DDS is to design the fragmentation and allocation of the underlying database. Fragmentation unit can be a file where allocation issue becomes the file allocation problem. The data allocation problem, is NP-complete, and thus requires fast heuristics to generate efficient solutions. Furthermore, the optimal allocation of database objects highly depends on the query execution strategy employed by a distributed database system, and the given query execution strategy usually assumes an allocation of the fragments.
A major cost in executing queries in a distributed database system is the data transfer cost incurred in transferring relations (fragments) accessed by a query from different sites to the site where the query is initiated. The objective of a data allocation algorithm is to determine an assignment of fragments at different sites so as to minimize the total data transfer cost incurred in executing a set of queries. This is equivalent to minimizing the average query execution time, which is of primary importance in a wide class of distributed conventional as well as multimedia database systems.
The rest of this paper is structured as follows: in section 2 we will introduce in static algorithms for allocating fragments, section 3 is about dynamic data allocation and section 4 contributes in transparent data relocation in dynamic environments in which settings are changed. Finally section 6 is our conclusion.

2. Static Algorithms

The data allocation problem is NP-complete in general and thus requires heuristics that are fast and are capable of generating high-quality solutions. Developing an efficient

heuristic highly depends on the query execution strategy employed by the distributed database system. This is because different query execution strategies have different data fragment migration patterns. A data allocation algorithm takes the following parameters as inputs: (i) the fragment dependency graphs, (ii) unit data transfer costs between sites, (iii) the allocation limit on the number of fragments that can be allocated at a site, and (iv) the query execution frequencies from the sites.
The fragment dependency graph models the dependencies between the fragments and the amount of data transfer incurred to execute a query. A fragment dependency graph (as shown in figure 1) is a rooted directed acyclic graph with the root as the query execution site (Site(Q) in figure 1) and all other nodes as fragment nodes (site(G), etc., in figure 1) at potential sites accessed by the query

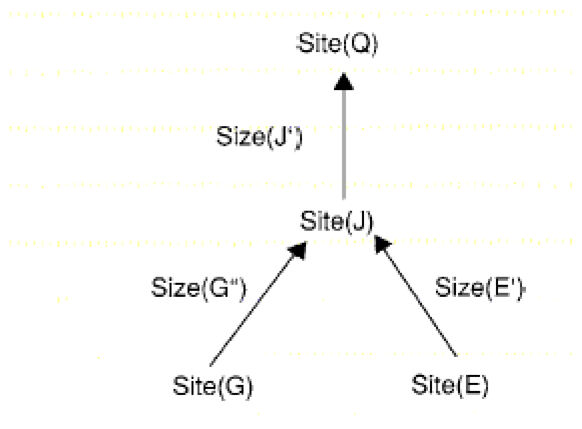


Figure 1 A fragment dependency graph (FDG)

In this section the proposed algorithms for data allocation problem will be described [2, 8, 9].

Generic Algorithm

Assuming that $r_{i,j}$ indicates the requirement by site i for fragment j, each fragment i is characterized by its size, si and ti,j indicates the cost for site i to access a fragment located on site j, the distributed database allocation problem is one of finding the optimal placement of the fragments at the sites. That is, we wish to find the placement, *P = {pt, p2, p3,..,pj, ...,pn}*(where pj = i indicates fragment j is located at site i) for the n fragments so that the capacity of any site is not exceeded, $\sum r_{i,j}\, s_j <= c_{i,j}\ \forall i \mid 1<=i<=m$
And the total transmission cost, $\sum\sum r_{i,j}\, t_{i,pj}$ is minimized [1].
By restricting the use of the requirements matrix and having zero transmission cost, the distributed database allocation problem can be transformed to the bin packing problem, which is known to be NP-complete.
A genetic algorithm (GA) is an adaptive search technique based on the principles and mechanisms of natural selection and ‘survival of the fittest' from natural evolution. By simulating natural evolution, in this way, a GA can effectively search the problem domain and easily solve complex problems. The GA begins by generating an initial

population, *P (t = 0)*, and evaluating each of its members with the objective function. While the terminator condition is not satisfied, a portion of the population is selected, somehow altered, evaluated, and placed back into the population.
The genetic algorithm for the data allocation problem is described below [2].

Genetic Data Allocation Algorithm:

```
(1) Initialize population. Each individual of the population is a concatenation of the binary representations of the initial random allocation of each data fragment.
(2) Evaluate population.
(3) No of generation = 0
(4) WHILE no of generation < MAX GENERATION DO
(5) Select individuals for next population.
(6) Perform crossover and mutation for the selected individuals.
(7) Evaluate population.
(8) No of generation ++;
(9) ENDWHILE
(10) Determine final allocation by selecting the fittest individual. If the final allocation is not feasible, then consider each over-allocated site to migrate the data fragments to other sites so that the increase in cost is the minimum.
```

It was found the GA to have superior performance to the greedy heuristic on fragment allocation problems of various sizes. While the greedy heuristic took time and effort to implement, the GA was very straightforward: an encoding was decided upon, and a simple function was written to evaluate candidate solutions.

The Simulated Evolution Algorithm

The principal difference between a genetic algorithm and an evolutionary strategy is that the former relies on crossover, a mechanism of probabilistic and useful exchange of information among solutions to locate better solutions, while the latter uses mutation as the primary search mechanism. Furthermore, in the proposed scheme the chromosomal representation is based on problem data, and solution is generated by applying a fast decoding heuristic (mapping heuristic) in order to map from problem domain to solution domain. The generic problem-space simulated evolution is as below:

```
(1) Construct the first chromosome based on the problem data and perturb this chromosome to generate an initial population;
(2) Use the mapping heuristic to generate a solution for each chromosome;
(3) evaluate the solutions obtained;
(4) no of generations = 0;
(5) WHILE no of generations < MAX GENERATION DO
(6) Select chromosomes for next population;
(7) perform crossover and mutation for these set of chromosomes;
(8) Use the mapping heuristic to generate a solution for each chromosome;
(9) evaluate the solutions obtained;
(10) no of generations = no of generations+1;
(11) ENDWHILE
(12) Output the best solution found so far;
```

Mapping heuristic: For each chromosome, we find a solution by allocating fragment *j* with the highest priority to the site *i* such that $u'_{i\,j}$ is smallest for all $u'_{x\,j}$, $1<x<m$. If the effective allocation limit embedded in the genes in part *a* of the chromosome for that site is exceeded (the site is already saturated), we allocate this fragment to the site with the next smallest value of u'_{ij}(the cost of allocating fragment j to site i) for all u'_{yj}, $1< y <m$, $y \neq x$. It is guaranteed that the fragment can be allocated to some site since we have checked that the total allocation limit is greater than or equal to the total number of fragments for every generation of chromosome. We then continue the process for the next fragment with the highest priority among fragments not yet allocated.

The Mean Field Annealing (MFA) Algorithm

Mean field annealing (MFA) technique combines the collective computation property of the famous Hopfield Neural Network (HNN) with simulated annealing. MFA was originally proposed for solving the traveling salesperson problem, as an alternative to HNN, which does not scale well for large problem sizes. It has been shown that MFA is a general approach that can be applied to various combinatorial optimization problems.

```
(1) Get the initial temperature $T0$, set $T = T0$.
(2) Initialize the spin averages $s = [s_{00}, s_{01}, . . . , s_{k-1,m-1}$, each $s_{i\,j}$ is initialized as a random number between 0 and 1.
(3) WHILE temperature $T$ is in the cooling range DO
(4) WHILE $E$ is decreasing DO
(5) Select a data fragment $i$ at random.
(6) Compute the mean field of the spins at the $i$ -th row, i.e., $\Phi_{i\,j}$, $\forall\, j$ .
(7) Compute the summation $\sum e^{\Phi ij/T}$
(8) Compute the new spin values at the $i$ -th row.
(9) Compute the energy change due to these updates.
(10) ENDWHILE
(11) Update the temperature $T$ according to the cooling schedule.
(12) ENDWHILE
(13) Determine the final allocation by allocating each data fragment to the site with the largest spin value. If the final allocation is not feasible, then consider each over allocated site to migrate the data fragments to other sites so that the increase in cost is the minimum.
```

Note that the last step of the MFA algorithm is necessary because we do not explicitly check for feasibility in the search process, which can then explore broader regions in the search space.

Random Neighborhood Search (RS) Data Allocation Algorithm

The main idea in a neighborhood search algorithm is to generate an initial solution with moderate quality. Then, according to some pre-defined neighborhood, the algorithm probabilistically selects and tests whether a nearby solution in the search space is better or not. If the new solution is better, the algorithm adopts it and starts searching in the new neighborhood; otherwise, the algorithm selects another solution point. The algorithm stops after a specified number of search steps have elapsed or the solution does not improve after a fixed number of steps. The solution quality of a neighborhood search technique relies heavily on the construction of the solution neighborhood. The algorithm for the data allocation problem is given as follows:

```
(1) Use Divisive-Clustering to find an initial allocation Initial Alloc;
(2) Best Alloc = Initial Alloc;
(3) New Alloc = Best Alloc; iteration = 0;
REPEAT
(4) searchstep = 0; counter = 0;
REPEAT
(5) Randomly select two sites from New Alloc;
(6) Randomly select two data fragments from each site;
(7) Exchange the two data fragments;
(8) IF cost is reduced THEN
adopt the exchange and set counter to 0;
ELSE otherwise undo it and increment counter;
UNTIL ++searchstep > MAXSTEP OR counter > MARGIN;
(9) IF cost(New Alloc) < cost(Best Alloc) THEN
Best Alloc = New Alloc;
(10) Randomly exchange two data fragments from two randomly selected distinct
sites from New Alloc; /* Probabilistic jump */
UNTIL iteration > MAXITERATION;
```

The SE and GA can generate better solutions than RS and MFA in general. However, the difference in solution quality is not always of the same magnitude. Also, the RS algorithm has the lowest time complexity which makes it the fastest algorithm, whereas the SE algorithm is quite slow. For fast running time and a small degradation of the optimality of the solution, the RS algorithm is a viable choice. On the other hand, when efficiency and solution quality are equally important, the GA may be a more attractive choice. Thus, the advantage of having these algorithms in a distributed database is the availability of a diverse range of solution quality and complexity trade-off.

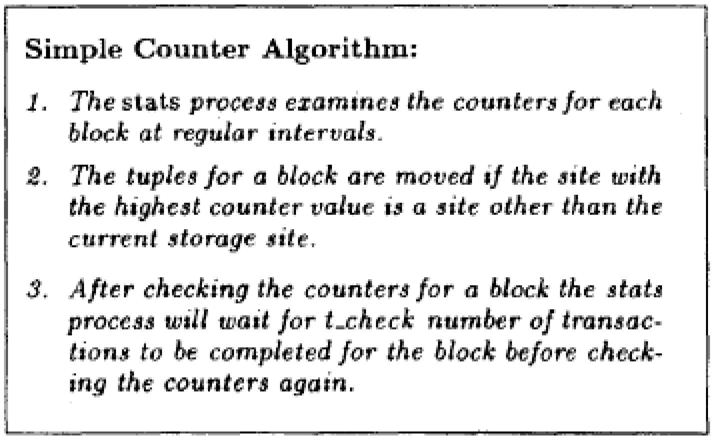
For fragment allocation design, in [10] a method is designed to meet the requirements of clustering sites and determining fragment allocation in distributed database system, minimizing the communication cost between sites, and enhancing the performance in a heterogeneous network environment system. Clustering method is developed to group the sites into clusters, which helps in reducing the communication costs between the sites during allocation process. Fragment allocation method is developed to enhance system performance by increasing availability and reliability where multiple copies of the same fragments are allocated.

3. Dynamic Allocation Algorithms

In the above studies, data allocation has been proposed prior to the design of a database depending on some static data access patterns and/or static query patterns. In a static environment where the access probabilities of nodes to the fragments never change, a static allocation of fragments provides the best solution. However, in a dynamic environment where these probabilities change over time, the static allocation solution would degrade the database performance. In this section different dynamic data allocation algorithms in distributed database systems are explained.

Simple Counter Algorithm

In this algorithm, in order to determine when a re-allocation is needed, it maintains weighted counters of the number of accesses from each site to each block. The counters for a particular block are maintained at only one of the sites in the system.
For example assume site 1 is the home site for partition A. In a system with N sites, site 1 would maintain N counters for partition A. Simple counter algorithm simply ranks the sites according to counter values and picks the best site. The simple counter algorithm is shown in Figure 2 [3].



Simple Counter Algorithm:

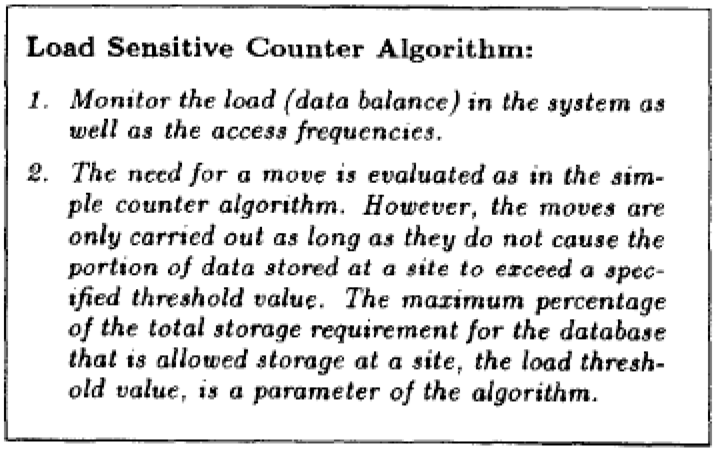
1. *The stats process examines the counters for each block at regular intervals.*
2. *The tuples for a block are moved if the site with the highest counter value is a site other than the current storage site.*
3. *After checking the counters for a block the stats process will wait for t_check number of transactions to be completed for the block before checking the counters again.*

Figure 2 The Simple Counter Algorithm

In this algorithm the state process is the process which accumulates statistics such as throughput, average response time for a transaction, and the fraction of transactions requiring access to remotely stored data. Note that the value of *t-check* should be small enough to allow the system to respond quickly to workload changes but large enough to prevent premature signaling of a change in access frequencies and having the data bounce back from a move soon after. Determining whether a data block needs to move is fully local decision, since all counters for a block are stored at the same site. This algorithm outperforms the static data allocation.

Load Sensitive counter algorithm

The simple counter algorithm works well as long as the load in the system remains low relatively balanced. However, in some cases, tuples for all blocks can end up at the same site, if most requests for all blocks originate at the same site. Although the fraction of local access is optimal, this allocation may cause poor performance due to disk or CPU overloading at the site. The load sensitive algorithm as shown in figure 3 [3] addresses this problem by taking into account load conditions; after all, if too many blocks were placed on a single machine then potential inter-transaction parallelism could be lost and overall throughput might decreases. This algorithm outperforms the simple counter algorithm as well as static allocation for a class of workloads.



Load Sensitive Counter Algorithm:

1. *Monitor the load (data balance) in the system as well as the access frequencies.*
2. *The need for a move is evaluated as in the simple counter algorithm. However, the moves are only carried out as long as they do not cause the portion of data stored at a site to exceed a specified threshold value. The maximum percentage of the total storage requirement for the database that is allowed storage at a site, the load threshold value, is a parameter of the algorithm.*

Figure 3 The Load Sensitive Counter Algorithm

Incremental Algorithm

[4] The incremental growth framework as shown in figure 4 is invoked when system performance, as computed using the objective cost function, is below the acceptable threshold (specified by DBA). To return to an acceptable state, new servers are introduced incrementally, one at a time, into the distributed database system. With the introduction of each new server, a new data reallocation for the system is computed. This process is iteratively executed until acceptable performance is achieved or the number of servers equals the number of relations in the distributed database system (the latter constraint can easily be relaxed in a distributed database system housing partitioned data).
In a distributed database system, an increase in workload typically necessitates the installation of additional database servers followed by the implementation of expensive data reorganization strategies. The incremental algorithm presents the Partial RELLOCATE and Full RELLOCATE heuristics for efficient data allocation. Both algorithms are greedy, iterative, “hill-climbing” heuristics that will not traverse the path twice. With each iteration, they will find a lower cost solution, or they will terminate. Both algorithms requires as input: the current data allocation, the relations, and the queries in the distributed database systems. Complexity is controlled and cost is minimized by allowing only incremental introduction of servers into the distributed database system. Due to their linear complexity, both heuristics ca be used to solve both small and large, complex problems, based on organizational needs. The algorithms reduce problem search space, and hence the cost of testing relation-server combinations.

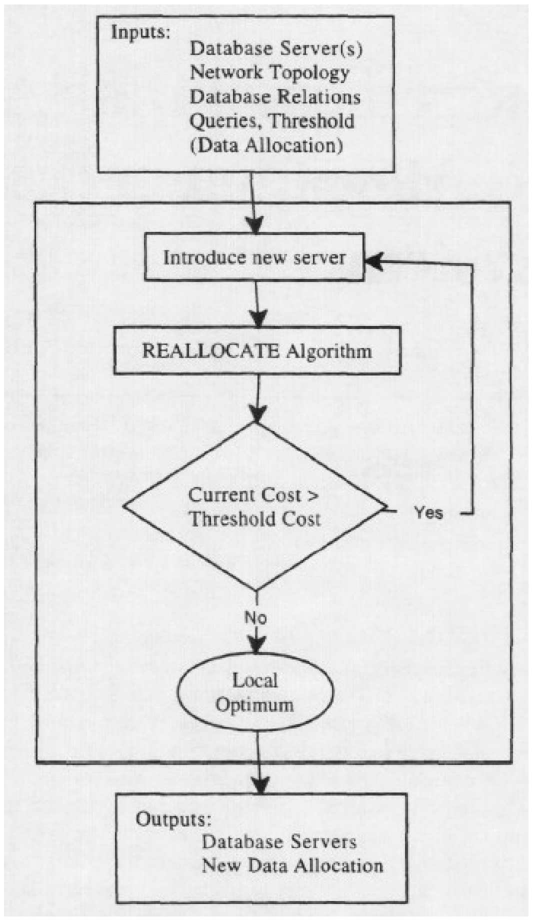


Figure 4 The incremental growth framework

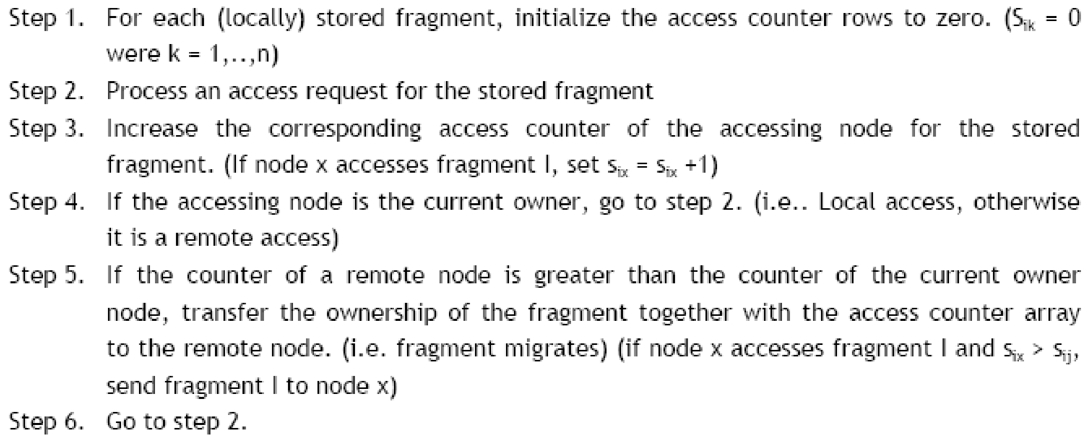
Threshold Algorithm

In this approach, horizontal, vertical or mixed fragmentation can be used. Allocation unit can even be as small as a record or an attribute.

As noted above, in distributed database systems, the performance increases when the fragments are stored at the nodes from which they are most frequently accessed. The problem is to find this particular node for each fragment. Counting the accesses of each node to a fragment offers a practical solution. Having the highest access value for a particular fragment, a node could be the primary candidate to store the fragment. An m by n access counter matrix S, where m denotes the number of fragments and n denotes the number of nodes, is shown below, in which Every element sij of S, where sij.Z^+.{0} (i.e. non-negative integers), shows the number of accesses to fragment I by node j.

$$S \bullet \begin{bmatrix} s_{11} & s_{12} & .. & s_{1n} \\ s_{21} & s_{22} & .. & s_{2n} \\ : & : & :: & : \\ s_{m1} & s_{m2} & .. & s_{mn} \end{bmatrix}$$

Initially, all fragments are distributed to the nodes according to any method. Afterwards, any node j runs the optimal algorithm given in Figure 5 for every fragment I, which it stores [5,7].



Step 1. For each (locally) stored fragment, initialize the access counter rows to zero. (S_{ik} = 0 were k = 1,..,n)

Step 2. Process an access request for the stored fragment

Step 3. Increase the corresponding access counter of the accessing node for the stored fragment. (If node x accesses fragment I, set $s_{ix} = s_{ix} + 1$)

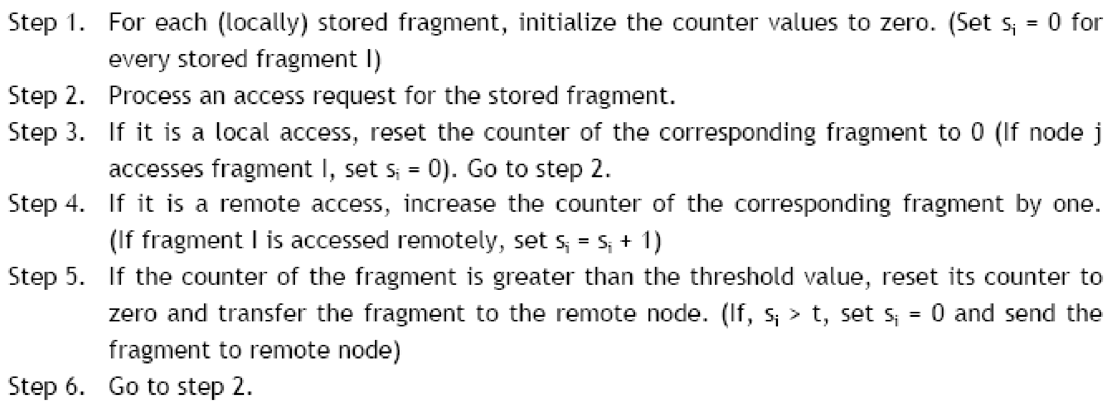
Step 4. If the accessing node is the current owner, go to step 2. (i.e.. Local access, otherwise it is a remote access)

Step 5. If the counter of a remote node is greater than the counter of the current owner node, transfer the ownership of the fragment together with the access counter array to the remote node. (i.e. fragment migrates) (if node x accesses fragment I and $s_{ix} > s_{ij}$, send fragment I to node x)

Step 6. Go to step 2.

Figure 5 Optimal Algorithm

An advantage of the optimal algorithm is the central node independence. That is, since each node runs the algorithm autonomously, there is no central node dependence. Every node is of equal importance. Whenever one node crashes, the algorithm may continue its operation without the fragments stored in the crashed node.
There are two drawbacks associated with the optimal algorithm. First one is the potential storage problem. As the fragment size decreases and/or the number of nodes increases, the size of access counter matrix increases, which in turn results in extra storage space need for the access counter matrix. The second drawback is the scaling problem for the data type that stores the access counter values. Since access counter values are continuously increasing, this problem may result in anomalies. A potential timing problem, which may cause back and forth migration of a fragment, deserves explanation.
In threshold algorithm, only one counter per fragment is stored. Comparing it to the optimal algorithm, this radically decreases the extra amount of storage space to just one value compared to an array of values in the optimal algorithm. There are two strategies to select the new owner. Either it is chosen randomly, or the last accessing node is chosen. In the former, the randomly chosen node could be one that has never accessed the fragment before. So picking the latter strategy is heuristically more reasonable.
Initially, all fragments are distributed to the nodes according to any method. A threshold value t is chosen. Afterwards, any node j, runs the threshold algorithm given in Fig. 6 for every fragment I, that it stores.



```
Step 1.  For each (locally) stored fragment, initialize the counter values to zero. (Set $s_i = 0$ for
         every stored fragment I)
Step 2.  Process an access request for the stored fragment.
Step 3.  If it is a local access, reset the counter of the corresponding fragment to 0 (If node j
         accesses fragment I, set $s_i = 0$). Go to step 2.
Step 4.  If it is a remote access, increase the counter of the corresponding fragment by one.
         (If fragment I is accessed remotely, set $s_i = s_i + 1$)
Step 5.  If the counter of the fragment is greater than the threshold value, reset its counter to
         zero and transfer the fragment to the remote node. (If, $s_i > t$, set $s_i = 0$ and send the
         fragment to remote node)
Step 6.  Go to step 2.
```

Figure 2 The Threshold Algorithm

An important point in the algorithm is the choice of threshold value. This value will directly affect the mobility of the fragments. It is trivial that as the threshold value increases, the fragment will tend to stay more at a node; and as the threshold value decreases, the fragment will tend to visit more nodes. Another point in the algorithm is the distribution of the access probabilities. If the access probabilities of all nodes for a particular fragment are equal, the fragment will visit all the nodes. The same applies for two nodes when there are two highest equal access probabilities.
In this algorithm, it is shown that the fragment tends to stay at the node with higher access probability. As the access probability of the node increases, the tendency to remain at this node also increases. It is also shown that as the threshold value increases, the fragment will tend to stay more at the node with higher access probability.

5. Transparent Relocation of Data

In a distributed system, long-running distributed services are often confronted with changes to the configuration of the underlying hardware. If the service is required to be highly available, the service needs to deal with the problem of adapting to these changes while still providing its service and make it transparent to client processes. This problem is increased further if multiple changes can occur concurrently. In [6] one of the approaches for data relocation is represented to address the problems of data relocation noted above as will be explain in this section.
In this approach, it is assumed that the service is implemented by a set of *server processes*, with each server located on a different machine and managing (or hosting) a disjoint subset of the records (assume no replication). Every record is always hosted by a single server, which is called the record's *hosting server*. The distributed service provides its services to external *client processes*.
The load distribution policy is captured by a data structure which is called the *mapping* which defines for each record its hosting server and can be used for redirecting requests to appropriate servers. Mappings are managed by a separate service, called the *configuration service* which is responsible for building an updated mapping that includes the configuration change and providing the new mapping to all the servers in the case of configuration change.
To invoke an operation, a client arbitrarily picks a server and submits its request to it. The selected server then looks in its copy of the mapping to determine the record's hosting server and

forwards the request accordingly. When a server receives the new mapping, it starts relocating records, includes transferring of mappings between a server and the configuration service, as well as relocation of records between servers.
The algorithm is proposed for multiple situations of data redistribution described as the following:

➢ **Single Redistribution:**
For single redistribution, the solution follows three steps:
● *Initialization*: Initially, all the servers have a local copy of the authoritative mapping, *M*, which is used for forwarding requests to the proper hosting. When the configuration service receives the notification for a configuration change, it computes a new mapping *M'* that reflects the change, and distributes *M'* to *all* the servers of the distributed service
● *Record relocation*: the server, receives a new mapping *M'* and ships its own record if it is needed. During the record relocation step, servers continue to forward client requests using the authoritative mapping *M* or *M'* for *not-yet-shipped* record and *already-shipped* record respectively. *Already-shipped* record requests are forwarded to the record's *new* hosting server as dictated by *M'*.
● *Termination*: As soon as a server completes its record relocation step, it notifies the configuration service. When the configuration service receives completion notifications from *all* servers, it, in turn, notifies all the servers that the termination step can start in which each server discards the mapping *M* and replaces it by the new mapping *M'* as the new authoritative mapping.
Different mappings are serviced sequentially. To make the record relocation step more efficient, a server does not discard a record after it has been shipped to its new host. Instead, the server keeps handling *lookup* requests for such a record, but only for as long as that record remains consistent with the copy at its new hosting server, it means before the first *update* request for that record is made.

Alternative Design Considerations

There are two strategies to deal with requests for already-shipped data while redistribution is in progress:
1. Reject the request and let the client keep trying until the redistribution is completed. It is contradictory of transparency goal.
2. Always handle the request locally independent of whether it is a lookup or an update request. In the case of an update request for a record that is already relocated, the authoritative hosting server propagates the record's modified value to its new hosting server in order to keep the two copies of the record consistent. This solution has the advantage that update requests are processed slightly faster, but introduces additional complexity for keeping the records consistent.

Also, there are two approaches for initial forwarding of requests:
1. Forwarding a request to the record's authoritative hosting server and having it forwarded further if the record is already shipped.
2. Forwarding the request to the record's new hosting server and having the record fetched on demand.

The first strategy favors frequent lookups and rare updates and the second strategy favors more frequent updates as it eliminates the extra forwarding of every single request for a shipped record that has been updated.

- **Overlapping Redistributions**

In the following approaches efficiency is improved by introducing concurrency (let R_1, R_2,.., R_n be the sequence of upcoming redistributions and M_1, M_2,. . . , M_n their respective mappings. M is the (current) authoritative mapping of the distributed service as a whole).

Approach I: Per-server Sequential Redistribution

In this case, the configuration service generates a new mapping and distributes it to the servers as soon as it receives a notification for a new configuration change. The servers themselves are responsible for locally queuing incoming mappings and processing them one at a time in the order received. Each server maintains a *queue of mappings*, which always contains *at least* one mapping. A server that has relocated all records for redistribution $R1$ can start carrying out the record relocation for the next redistribution $R2$ before all other servers have completed redistribution $R1$. The authoritative mapping as known to the server (i.e., M) is removed from the server's queue only upon receiving a notification from the configuration service stating that redistribution $R1$ has been completed by all servers. The forwarding is based on *first* preference virtual mapping.

Approach II: Per-server Mixed but Ordered Redistributions

The main idea in this case is that there are cases where a server does not need to complete a redistribution to start working on the next one. Assume a server is currently going through its set of records, checking which ones are to be shipped based on redistribution Ri and it comes across a record that is not remapped by Ri. The server can then ship this record based on a successive redistribution $Rj(j > i)$, even if it has not finished Ri yet. As the approach (I) forwarding requests are based on *first* preference virtual mapping.

Approach III: Direct Shipping to Final Destination

The optimization introduced in Approach III entails that a record is shipped directly to the record's hosting server according to the *last* known redistribution. This policy keeps a record from being shipped from server to server when it is already known that it needs to be shipped further. Instead, the record is sent directly to the last server in the chain of servers it is mapped to. This policy prevents unnecessary network traffic and redistribution delay.
The main difference between this approach and approaches I and II is that server ships records based on the virtual mapping with *last* preference of all the mappings in its queue. The records are thus directly relocated to the proper hosting server. However, servers still use the virtual mapping with *first* preference of all these mappings to forward requests that cannot be handled locally.

6. Conclusion

In this paper, we address the problem of non-redundant data allocation of fragments in distributed database system. We have also described various proposed algorithms, designed based on the evolutionary computing paradigm, for the data allocation problem in distributed database systems.

We have described and compared the four proposed data allocation heuristics, including a genetic algorithm (GA), a simulated evolution (SE) algorithm, a mean field annealing (MFA) algorithm, and a random search (RS) algorithm. We also contributed in different dynamic data allocation algorithms (Simple Counter, Load Sensitive Simple Counter, Incremental, and Threshold algorithms) and compared them.

For dynamic data allocation the transparent data relocation is needed. So, this paper deals with a management issue of distributed services, namely the redistribution of non-replicated data among the servers comprising a distributed service. The objective has been to redistribute the data without disrupting the service's availability. The main contribution of this subject is to show that transparent data redistribution is possible. That is, it is possible to carry out such redistribution in a way that is totally transparent to clients of the service.

7. References:

[1]L. C. John, A Generic Algorithm for Fragment Allocation in Distributed Database Systems*, ACM*, 1994

[2] Ahmad, I., K. Karlapalem, Y. K. Kwok and S. K. Evolutionary Algorithms for Allocating Data in Distributed Database Systems, *International Journal of Distributed and Parallel Databases*, 11: 5-32, The Netherlands, 2002.

[3] A. Brunstroml, S. T. Leutenegger and R. Simhal, Experimental Evaluation of Dynamic I)ata Allocation Strategies in a Distributed Database with changing Workloads, *ACM Transactions on Database Systems*, 1995

[4] A. G. Chin, Incremental Data Allocation and ReAllocation in Distributed Database Systems, *Journal of Database Management;* Jan-Mar 2001
; 12, 1; ABI/INFORM Global pg. 35

[5] T. Ulus and M. Uysal, Heuristic Approach to Dynamic Data Allocation in Distributed Database Systems, *Pakistan Journal of Information and Technology 2 (3): 231-239*, 2003
, ISSN 1682-6027

[6] S. Voulgaris, M.V. Steen, A. Baggio, and G. Ballintjn, Transparent Data Relocation in Highly Available Distributed Systems. *Studia Informatica Universalis*. 2002

[7] Navathe, S.B., S. Ceri, G. Wiederhold and J. Dou, Vertical Partitioning Algorithms for Database Design, *ACM Transaction on Database Systems*, 1984
, 9: 680-710.

[8] P.M.G. Apers, “Data allocation in distributed database systems,” *ACM Transactions on Database Systems*, vol. 13, no. 3, pp. 263–304, 1988.

[9] Y. F. Huang and J. H. Chen, Fragment Allocation in Distributed Database Design *Journal of Information Science and Engineering 17*, 491-506, 2001

[10] I. O. Hababeh , A Method for Fragment Allocation Design in the Distributed Database Systems, *The Sixth Annual U.A.E. University Research Conference*, 2005

s