The Impact of Workload Clustering on Transaction Routing

Christos Nikolaou[a] Alexandros Labrinidis[b] Volker Bohn[c] Donald Ferguson[d]
Michalis Artavanis[a] Christos Kloukinas[a] Manolis Marazakis[a]

Abstract

The qualitative and quantitative description of the workload of a system is very important for capacity planning and performance management. In large-scale transaction processing systems, dynamic workload control algorithms are applied to optimize system performance. Such algorithms can benefit from the results of workload clustering algorithms that partition the workload into classes consisting of units of work exhibiting similar characteristics. This paper presents *CLUE*, a clustering environment for OLTP workload characterization. *CLUE* provides a library of clustering algorithms that classify transactions into classes, according to their database reference patterns. This paper introduces *HALC*, a new batch-mode heuristic clustering algorithm, designed to cope with the large volume of input data that is typical for real-life applications. Next, an *on the fly* clustering algorithm based on neural networks is described. This algorithm can be used in an on-line fashion in systems whose characteristics change through time. This paper provides an evaluation of the performance of *HALC* and the *on the fly* algorithms in terms of execution times and statistical metrics related to the quality of clusters that they compute, for both synthetic and real-life workload traces. Finally, this paper quantifies the impact of workload clustering on the performance of three dynamic transaction routing algorithms for Shared-Nothing transaction processing systems.

1 Introduction

Typical multi-node on-line transaction processing (OLTP) systems employ a shared-nothing architecture where each individual node has private disk & main memory and any communication among nodes is done through the local area network. Data are usually placed statically on nodes, and remote access is needed when a transaction has to access data that do not reside in the execution node's disk.

Knowledge of the intrinsic characteristics of the transaction workload is essential for dynamic workload control algorithms that are used in multi-node OLTP systems to optimize performance. Examples of such intrinsic characteristics, which are independent of the arrival rates of units of work, include the average number of database accesses, the files accessed, CPU processing demands, and the average number of synchronization points. Large-scale OLTP systems provide a controlled run-time environment for predefined transaction programs ("canned transactions") that model certain business actions in the application domain.

[a]Department of Computer Science, University of Crete, Greece and Institute of Computer Science (ICS), Foundation for Research & Technology - Hellas (FORTH) Heraklion, Crete, Greece. Email: nikolau@ics.forth.gr, artav@ics.forth.gr, kloukin@csd.uch.gr, maraz@ics.forth.gr

[b]Department of Computer Science, University of Maryland, USA. Email: labrinid@cs.umd.edu

[c]Hamburg-Mannheimer, Abt. EDVS/Ueberseering 45/D-22297, Hamburg, Germany. Email: bohn@acm.org

[d]IBM T. J. Watson Research Center, USA. Email: dfferg@watson.ibm.com

A classic example is the *debit/credit* transaction in the banking application domain. Users submit requests specifying the type of transaction to execute together with any required parameters (such as the account number and amount of money in the debit/credit example) and the transaction monitor activates the appropriate predefined program with these parameters.

An enterprise's geographical and organizational structure has a strong influence on the placement and access to operational data; certain users will only use a predefined set of terminals to access specific data, which are related to their daily work and the location where they work. For the debit/credit example, bank tellers require access to bank accounts that are associated with their branch. It is expected that the combination of the transaction program name, user identification, and terminal name can be used as a unique identifier of transactions that will most likely display similar intrinsic resource consumption characteristics. The combination of these three names, hereafter called a *triplet ID*, serves as a descriptor for all units of work that share this combination. Units of work emanate from the requests of users. The *triplet ID* together with statistical data that describe the expected resource consumption demands of its corresponding units of work is called a *triplet*. Large-scale OLTP systems can use as many as 1000 canned transactions and support more than 100,000 terminals and users [25]. Though the real number of *triplet IDs* that occur in a running application is much smaller than the product of the number of transaction programs, users and terminals, it could easily be in the order of tens of thousands.

Given a description of intrinsic workload characteristics for *triplet IDs*, it is essential for the success of dynamic workload control algorithms to compress this data without losing information necessary for effective workload control. This means that triplets with similar resource demands and database referencing behavior have to be identified and grouped into classes with similar characteristics. Triplets with different characteristics must be assigned to different classes. This assignment of triplets to classes, hereafter referred to as *workload clustering*, is usually done by an expert user (e.g. a database administrator), through a series of empirical decisions. However, this approach is not realistic for large-scale transaction processing systems. Automatic clustering techniques have to be adopted. The intrinsic characteristics of clusters provide a characterization of the workload from which the clusters were created. Thus, workload clustering provides valuable input to dynamic transaction routing algorithms that are responsible to assign each incoming unit of work to a processing node of the OLTP system.

This paper presents *CLUE*, a clustering environment for OLTP workloads, which provides a library of clustering algorithms. These algorithms operate on traces collected from operational OLTP systems. We introduce *HALC*, an efficient batch-mode algorithm that can handle clustering of large volumes of data. *HALC* has been integrated in the *CLUE* environment. This paper compares the performance of the clustering methods provided by *CLUE* for several synthetic traces as well as of two real-life traces produced by an OLTP system based on a centralized relational database system, in terms of execution times and statistical metrics related to the quality of the clusters that they compute. Furthermore, a major contribution of this paper is to provide an understanding of the impact of workload clustering in the performance of OLTP systems. We use *TPsim* [19], a transaction processing system simulator to study the performance of three different dynamic transaction routing algorithms for each of the clustering algorithms compared in this paper.

The paper is organized as follows. Section 2 provides a brief survey of work related to workload clustering. Section 3 describes the *CLUE* environment and section 4 presents the *HALC* batch-mode algorithm. Section 5 presents an *on the fly* clustering algorithm, based on neural networks, that has also been incorporated into *CLUE*. An evaluation of the quality of the afore-mentioned clustering methods is presented in Section 6, while Section 7 provides

the results of simulation experiments that were conducted with *TPsim* to study the degree of usefulness of workload clustering to the overall system performance. Finally, concluding remarks appear in Section 8.

2 Related Work

This section provides a brief overview of related work in the areas of clustering and workload characterization.

There is a large collection of clustering algorithms which are used to analyze experimental data in a variety of applications, and new algorithms continue to appear in the literature. Cluster analysis is the process of classifying objects into subsets that have meaning in the context of a particular problem. A classification of the most widely used methods for clustering is given in [15]. Automatic clustering techniques are valuable tools for workload classification, because they eliminate the dependency of classification results on the analyst's skills and subjective judgments. However, as commented in [24], automatic clustering techniques should be combined with expert knowledge about the case being studied, in order for the clustering results to be valid. The major problems of the general approach of workload clustering are discussed in [11]. These are: the selection of the clustering algorithm and of the clustering distance metric, the appropriate scaling and normalization of the input data, and the efficient handling of large sized inputs.

The behavior of real workloads is very complex and difficult to reproduce. A model for such workloads has to capture the static and the dynamic behavior of the real load and must be compact, repeatable and accurate [4]. A discussion of the problems arising when one tries to generate such models can be found in [2]. The measurement-based approach to workload characterization is addressed in [3]. In this work, various types of systems, namely interactive systems, distributed and parallel systems, and databases, are analyzed and a number of case studies are presented.

In studies on relational database workloads, measurements are collected in order to perform workload characterization. The measured events dealt mainly with buffer manager I/O, lock information, and SQL statements distribution and composition. For example, it is shown (in [16] - with analysis of a database trace) that general models for program traces are not applicable for detailed database workload characterization. [29] presents a *Relational Database Workload Analyzer*, which aims at characterizing the workload in a DB2 environment. The authors performed a workload study, focusing on the structure and complexity of SQL statements, the makeup and run-time behavior of transactions/queries, and the composition of relations and views.

[30] and [31] propose grouping transactions according to their consumption of system resources, and focus particularly on *affinity clustering*, i.e. partitioning the transactions into clusters according to their database reference patterns. In [31] data placement and transaction clustering are determined by solving an optimization problem where the objective function takes into account the balance of processing load by the affinity clusters produced. An analytical modeling approach is followed to examine the impact of affinity clustering to the performance of the Shared Disk, Shared Intermediate Memory and Shared Nothing architectures.

In this paper, we assume that the placement of data has already been determined, and instead focus on the problem of workload characterization through cluster analysis. We quantify the actual impact of different clustering algorithms on the performance of an OLTP system by examining how the workload characterization estimated from cluster analysis affect the behavior of dynamic transaction routing algorithms.

3 Workload Clustering with *CLUE*

We first give a top level description of *CLUE* and then briefly describe the clustering algorithms that were implemented.

3.1 Description

CLUE performs clustering of the input triplets into utilization classes in three steps:

Read Triplets into Main Storage: The input file is produced by a trace filtering and correlation tool. This tool reads trace data derived from a running OLTP application, correlates the various types of trace records to triplets and generates the triplet data to be read by the clustering algorithm. The input file, derived from the original trace file that contains one entry for each data access made by a transaction, contains summary information about the total number of user IDs, terminal IDs, and transaction IDs, the number of file /database pages accessed, and the number of triplets. The largest portion of the input file contains data describing the page references made by each triplet, and whether these references were read or update requests. Currently, only the information relating to the number of page references made by each triplet is taken into account by *CLUE*.
Each triplet is described by the following information:

TRIPLET_ID:
TRAN_ID, USER_ID, TERM_ID
GLOBAL INFORMATION:
Number of SYNCPOINTS,
CPU burst between data requests,
Number of page references (data requests) ("transaction length"),
Read-only indicator (true, if triplet submitted read accesses only),
Name information (name conventions used in various applications)
ACCESS INFORMATION:
Number of read references to each data file,
Number of write references to each data file.

The information from the input file is stored in a *reference matrix* (see fig. 1), where the rows are indexed by the triplets and the columns by the database page references. That is, in row i and column j lies the element referring to triplet i and data page j. Each matrix entry contains two fields, giving the number of read and write references to the data page associated with the triplet. This matrix is expected to be too large to be stored in its entirety in main memory, therefore an efficient representation method is required. Since experimentation with real-life traces showed that the *density* of this matrix, defined as the ratio of non-zero cells to the total number of cells, is below 1%, a sparse matrix provides an efficient representation of the *reference matrix*.

Preprocessing of the input: During the preprocessing phase, *CLUE* may perform various numerical transformations to the information related to each triplet, as well as change the relative order of the triplets (which for some algorithms may provide a better initial state). The various alternatives, which may be specified by the user in a configuration file, are:

Scaling and handling of the outliers: Scaling is done using either the classical standardization or the logarithmic transformation, as described in [24]. This is necessary, since

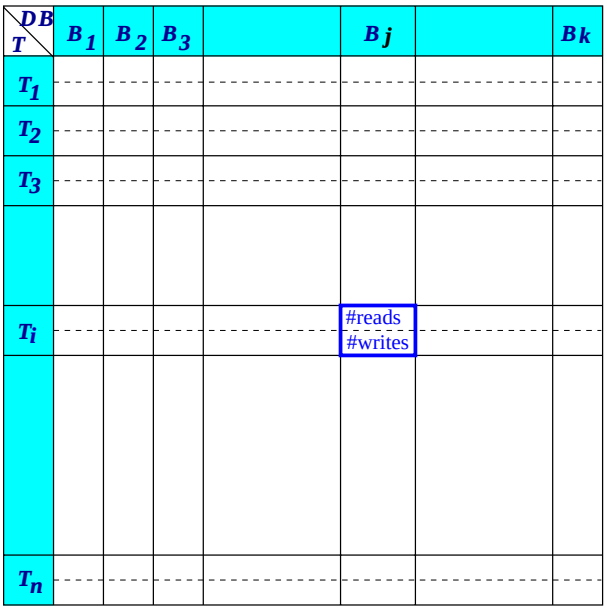

Figure 1: Triplet-page reference matrix.

an improper scaling and treatment of the outliers usually leads to bizarre clustering results.

Sorting of the triplets: This phase attempts to sort the triplets, so that triplets referencing the same pages will be placed as close as possible in the reference matrix. The sorting algorithm used in the current implementation is Heapsort and the criterion for the sorting can be one of the following:

REF sorts triplets according to the total number of page references they make

LEX sorts the triplets lexicographically. That is, it considers each page to be a letter and the "word" constructed for each triplet contains the respective letter if the triplet made any references to that page.

BEA sorts the triplets as in the first phase of the Bond Energy algorithm [21], where a function *bond()* is computed for each pair of triplets, showing the degree of similarity of the page references they make. The larger the bond, the closer the two triplets will be placed, once the sorting is completed.

Clustering: After preprocessing the elements of the reference matrix, one may select among a variety of different clustering algorithms to partition the triplets into a number of utilization classes (classes consisting of transactions with an affinity to a particular database partition). The maximum number of classes that are to be produced should be specified by the user. In the experiments that were conducted with real-life traces, the maximum number of utilization classes was set to 100, because, in our experience, a reasonable number of utilization classes for transaction routing algorithms is usually less than 100. The result of the clustering, namely the utilization classes, is stored in a file so that another module of the OLTP system, such as a transaction router, can use it.

3.2 Implementation

Four batch-mode algorithms have been implemented: an algorithm that randomly assigns triplets to classes, the *K-Means* algorithm [18, 13, 1, 28], the Bond Energy algorithm [21], and the *HALC* algorithm which is presented in detail in section 4. The random algorithm was implemented to be used as a basis for comparison.

The *K-Means* algorithm is a well-known clustering algorithm and, as noted in [24], it is considered as the standard technique for computing workload classes from measurement data, since it is simple and quite fast. A quick description of this algorithm is the following: Initially the algorithm assumes a random partition of the data into K clusters. Then, it goes through a series of iterations. In every iteration, each datum is assigned to the cluster whose center is closer to it (according to some *distance metric*, e.g. Euclidean distance). The cluster centers are recomputed after each iteration and the iterations stop when the cluster memberships stabilize.

The third clustering algorithm, Bond Energy algorithm (*BEA*), performs very well when the number of data to be clustered is small, but its high complexity makes it unsuitable for use in any other case. Nevertheless, it was implemented in the hope it would exhibit good clustering results. Finally, the *HALC* algorithm is a heuristic algorithm which is very efficient in the case of large sized inputs. *HALC* will be described in more detail in section 4.

All the afore-mentioned clustering algorithms have been implemented to work with two different distance metrics, of the general form $\alpha(x, y)$:

- The *Pseudo Linear Dependency Metric*:

$$\alpha(x,y) = \max_{i=1,\ldots,n} \left\{ \begin{array}{ll} 0 & \text{if } x_i = y_i = 0 \\ \frac{\|x_i - y_i\|}{\max(x_i, y_i)} & \text{otherwise} \end{array} \right\} \tag{1}$$

  where n is the number of all data pages, x_i is the number of accesses that triplet x makes to data page i and y_i is the number of accesses that triplet y makes to the same data page i.

- The *Vector Distance Metric*:

$$\alpha(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2}$$

With the vector distance metric, each triplet is considered to be a vector in an n-dimensional space, where each dimension maps to a separate data page.

The number of accesses x_i used in the calculations of distances (eq. 1 and eq. 2), is a weighted sum of the read and write accesses for the corresponding triplet and database page, i.e. $x_i = \frac{w_{\text{read}} * \#\text{read accesses} + w_{\text{write}} * \#\text{write accesses}}{\frac{w_{\text{read}} + w_{\text{write}}}{2}}$. The weights w_{read} and w_{write} are specified by the user; their default value is 1 (similarly for y_i).

4 HALC - A Heuristic Algorithm for Clustering

This section provides an overview of the *HALC* algorithm, short for a Heuristic ALgortihm for Clustering. *HALC* is an agglomerative, partitional, clustering algorithm[1]

The algorithm works by initially assigning each triplet to a separate cluster. Then, in each iteration, the algorithm merges clusters with similar page references. Clusters are considered

[1] According to the classification scheme given in [15]

to be similar, if the distance between them is less than the current threshold value of the clustering distance. The clustering distance is set to a default value at first and is automatically adjusted at each iteration. The iterations stop when the number of clusters becomes equal to the number of clusters that the user specified, or when any valid change to the clustering distance does not affect the number of clusters. When two clusters X and Y are merged, there are two choices for computing the page references of the new cluster X'; they are set either to the sum of the references of X and Y, or to the weighted average of them, according to what the user had specified.

Each iteration consists of the following two phases, which are executed consecutively until the algorithm terminates:

Elementary Clustering Step (*ECStep*): Assuming an initial good sorting pass, triplets with similar characteristics must have been placed close to each other in the reference matrix (see fig. 1). During this step, the triplets are split into *intervals*, roughly as many as the requested number of final clusters. Intervals are used to limit the scope of distance calculation between triplets. The relative order of clusters is maintained and the length of each interval is computed assuming a base length equal to $\frac{\#\text{current clusters}}{\#\text{final clusters}}$ and a random negative or positive deviation.[2]

- Within each of the produced intervals, the distance between each element Y_i and the first element, Y, of the interval is computed.
- If for some pair (Y, Y_i) the distance is less than the current distance, Y_i is merged with Y. Then, Y_i is removed from the list of clusters.

This Elementary Clustering Step can be performed either in *simulation mode*, where no changes in the data structures are performed and simply the number of possible new clusters is returned, or in *normal mode*, where the clusters are merged and the result is irreversible.

Re-Adjustment of the Clustering Distance If Necessary (*RACDIN*): This step computes the decrease of the number of clusters, due to the last *ECStep*, and checks if it satisfies the following two criteria:

Quality criterion - The decrease should not be *larger* than a user specified percentage (e.g. 10%), or else the quality of the produced clustering might be poor.

Quantity criterion - The decrease should not be *smaller* than a user specified percentage k% of an *initial* decrease, or else it is too small and a lot of iterations will be needed in order to reach the required number of clusters.

There may be various choices for what can be selected as the *initial* decrease. It could be the decrease after the first iteration of the algorithm, the maximum or the average decrease observed so far, or even a user specified value.

If the quality criterion is not met, the clustering distance should be decreased. At this point, the algorithm goes through a loop, where in each iteration the clustering distance is decreased by a specified percentage and then an *ECStep* is performed in simulation mode, to check if the produced decrease satisfies the quality criterion. If the quality criterion is met or the maximum number of iterations is reached, the loop is exited, in which case the clustering

[2]The reader should note the difference between an interval and a cluster; an interval usually *contains* more than one clusters, which may have only one element (i.e. a simple triplet) or more elements.

distance is adjusted to its new value. If the quantity criterion is not met, the clustering distance should be increased and the algorithm follows a similar procedure as before, in order to adjust the clustering distance.

The quantity criterion is checked immediately after the quality criterion has been met. However, after each increase in the clustering distance, performed in an effort to meet the quantity criterion, the quality criterion is checked again. The algorithm must guard against oscillations (a decrease followed by an increase, or vice versa) that might occur. In such a case, depending on the option settings provided by the user, it either exits the *RACDIN* phase, or halves the decrease or increase percentage used (therefore allowing changes at a finer level of detail) and tolerates a few more oscillations before exiting. After an oscillation is detected, the user has various choices for the selection of the new clustering distance (the average of the encountered distances, the last distance, etc.)

It is obvious from the above description, that there are various parameters affecting the behavior of the *HALC* algorithm. Even though there are default values for all the parameters described above, a user who has knowledge of the nature of the data and the distances produced by the metrics used in the *CLUE* environment could select better values for these. In a future version, *HALC* should also try to estimate suitable values for these parameters automatically.

Currently, *HALC* takes into account only the information about the page references made by each triplet. However, more information should be considered for the generation of the utilization clusters; this was referred to as *global information* in section 3. This additional information can be used to further constrain the formation of clusters, in the following way: Only if the metric used to compute the clustering distance allows the merging of two clusters, *and* the number of synchronization points are similar, *and* ..., *and* the application names are similar, are the clusters themselves considered similar, and subsequently merged into one. Initial experiments have shown that these additional constraints can improve the clustering quality.

5 On the Fly Workload Clustering

So far, we have considered systems with workload characteristics that are, more or less, stable through time. If, however, workload clustering is to be extended to systems with characteristics that change through time, such as a banking transaction processing system where people may perform different transactions depending on different days of the month, then the batch-mode clustering algorithms are no longer sufficient. In such cases, there is a need for an *on the fly* clustering algorithm, which can follow the changes in the state of the system, and the type of user requests.

Such an algorithm should be fast, so that it does not degrade the performance of the system, accurate, and preferably simple. Artificial neural networks have all these properties. An artificial neural network, called *Optimal Adaptive K-Means* [6], was constructed for this purpose and was tested along with the other algorithms, in order to evaluate its performance.

Optimal Adaptive K-Means is an enhanced version of the neural network implementation of *K-Means*, which is usually referred to as *Adaptive K-Means* [17, 22, 14]. First, a brief description of *Adaptive K-Means* will be given, followed by a presentation of the *Optimal Adaptive K-Means*.

5.1 Adaptive K-Means

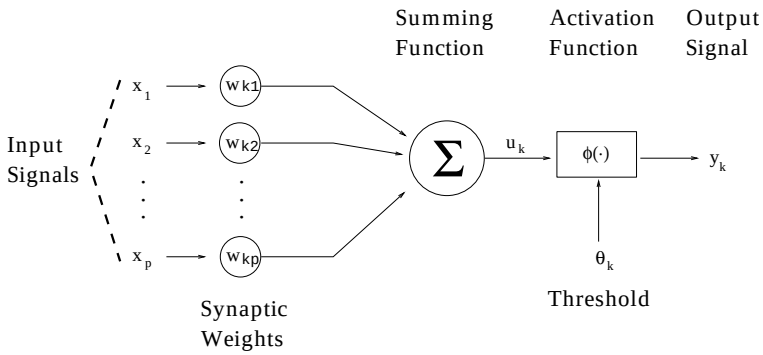

Figure 2: Mathematical Neuron Model

A mathematical neuron, or just neuron from now on, can be modeled as shown in Figure 2. In brief, a neuron k, computes a weighted sum of p input signals x_j, j = 1 ... p and generates a signal u_k usually between [0,1]. On this signal an activation function $\varphi(\cdot)$ is applied, along with a certain threshold θ_k. The final output signal can take two different values (0 or 1) depending on the activation function (which is usually a unit step function). Mathematically,

$$u_k = \sum_{j=1}^{p} w_{kj} x_j \tag{3}$$

$$y_k = \varphi(u_k - \theta_k) \tag{4}$$

where $x_1, x_2, \ldots, x_p$ are the input signals, $w_{k1}, w_{k2}, \ldots, w_{kp}$ are the synaptic weights of neuron k, u_k is the output of the summing function, θ_k is the threshold, $\varphi(\cdot)$ is the activation function and y_k is the output signal of neuron k.

The *Adaptive K-Means* neural network consists of K neurons, each one corresponding to a different cluster. The centroid of each cluster is stored in the synaptic weights vector, $\vec{c}$, of the respective neuron.

These weights are usually initialized with small random values and then updated in the following manner: As a new datum, $\vec{x}$, is presented to the network, all neurons compute the Euclidean distance between it and their centroid. The neuron having the smallest distance is said to be the *winner*, i.e. the one that represents the cluster that will "own" datum $\vec{x}$. Then, the winning neuron updates its synaptic weight vector, so that its centroid will move closer to the new data. The update is done accordingly to eq. 5, which is a running average of the data seen so far. The coefficient $M^{(i)}(\cdot)$, called the *membership indicator*, is calculated as in eq. 6 and is nothing more, than the mathematical expression "choose the neuron which is closer to the data". $d(\vec{\alpha} - \vec{\beta})$ denotes the squared Euclidean distance of $\vec{\alpha}$ and $\vec{\beta}$ (eq. 7). The coefficient η, appearing in eq. 5, is called the *learning rate* of the network and controls the speed and accuracy with which the network will converge to the final clusters[3].

$$\vec{c}^{(i)}[T+1] = \vec{c}^{(i)}[T] + M^{(i)}(\vec{x}[T]) * \eta * (\vec{x}[T] - \vec{c}^{(i)}[T]) \tag{5}$$

$$M^{(i)}(\vec{x}) = \begin{cases} 1 & \text{if } d(\vec{x} - \vec{c}^{(i)}) \leq d(\vec{x} - \vec{c}^{(j)}) \ \forall j \neq i \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

[3] η is usually a constant, in the [0, 1) interval. Value 0 means that the network has stopped learning, i.e. it has reached a stable state.

$$d(\vec{x}, \vec{c}^{(i)}) = ||\vec{x} - \vec{c}^{(i)}||^2 \tag{7}$$

The afore-mentioned Euclidean distance and membership indicator force the network to cluster data in such a way, that the *mean squared error* function, or *MSE* for short, decreases [14] [22]. This function is shown in eq. 8. Symbol $v^{(i)}$ in eq. 8 denotes the intra-cluster variance of the i-th cluster, i.e. the mean value of the distances of the data of the cluster from the centroid of the cluster, as shown in eq. 9 ($S^{(i)}$ being the number of elements of the i-th cluster, $\vec{x}_j^{(i)}$ the j-th datum of the i-th cluster, and $\vec{c}^{(i)}$ the centroid of the i-th cluster).

$$\text{MSE}(K) = \sum_{i=1}^{K} v^{(i)} \tag{8}$$

$$v^{(i)} = \frac{1}{S^{(i)}} * \sum_{j=1}^{S^{(i)}} ||\vec{x}_j^{(i)} - \vec{c}^{(i)}||^2 \tag{9}$$

5.2 Shortcomings of the Adaptive K-Means

The *Adaptive K-Means* algorithm has certain inherent shortcomings, that do not always permit it to reach the globally optimal solution. As a globally optimal solution we define the minimum square error for the entire clustering (see eq. 15) for a given data set.

First of all, some of its neurons may get initialized with values, that lie in regions of the search space with few or even no data at all. The effect of this is that these neurons will almost never win over the others, which means that the network will be using only a subset of its neurons and, therefore, it will have to join dissimilar clusters together. It is evident that in such a case the produced solution will be sub-optimal.

Various techniques have been proposed to remedy this problem; in [26, 27], it is proposed that the so called *leaky learning rule* be employed, according to which, non-winning neurons have their weights updated as well, but with a smaller learning rate than the winner, so that the later still retains an advantage over the others. In [9], it is proposed to change the distance function used, so that it favors those neurons which have won fewer times during the past. This later enhancement is called the *conscience learning law*. However, both these methods cause the cost function of the neural network (eq. 8) to change in such a way, that the clusters produced by the network are no longer optimal with respect to *MSE*, or, sometimes, not even near optimal. In addition to that, the *leaky learning rule* increases by far the computational complexity of the algorithm, since the synaptic weights of every neuron, instead of just the winner, must be recomputed at each step, when it is employed.

Another problem of the *Adaptive K-Means* algorithm is the fact that the learning rate is a constant. When choosing a suitable value for it, there is a tradeoff between the dynamic performance of the network, i.e. the rate by which it converges to the final solution, and the steady-state performance, i.e. the deviation of the final solution from the optimal solution. In other words, one must choose a small enough value, so that the network converges to a final solution and does not wander aimlessly in the search space. The smaller the learning rate is, the smaller the final deviation from the optimal solution will be. On the other hand, the smaller the rate is, the longer it takes for the network to reach a solution.

Since the optimal value for the learning rate depends on the characteristics of the data that are to be clustered, it is not possible to select one a priori. Usually, a small value is selected initially, which is then successively increased through a series of "test and fail" experiments. Several different methods have been proposed to alleviate this problem, by adjusting the learning rate dynamically. In [7], it was proposed that the learning rate be

inversely proportional to the square root of the number of data seen up to that point in time. Since this rule increases the time needed by the network to converge to the final solution, another method called *search then converge* [8] was later proposed by the same authors, where $\eta(t) = \frac{\eta_0}{(1+t/\tau)}$. In this method, the learning rate is kept close to the value η_0 for a time period equal to τ and then it starts decreasing at a rate of $\frac{1}{t}$. In many cases, this method converges both accurately and quickly. It is not possible, however, to select the optimal value for τ a priori. In addition to that, such methods are not suitable for problems whose characteristics keep changing during time.

Both of these problems, that of the exclusion of neurons due to bad initialization of their weights, and that of the determination of a good learning rate, were taken into consideration in the creation of the *Optimal Adaptive K-Means* algorithm [6], which we describe in the next section.

5.3 Optimal Adaptive K-Means

Optimal Adaptive K-Means is based upon the findings of Gersho [12], who showed that, for a large number K of clusters and a smooth underlying probability density function P of the data, all regions in an optimal *Voronoi partition* have the same within-region variations $v^{(i)}$. Since *K-Means* (and therefore *Adaptive K-Means*) produces such a Voronoi partition, the authors of [6] conjectured that it would be interesting to aim for a clustering with equal cluster variances, even when K is not large enough, or P is not smooth. In order to achieve this, they proposed two changes to the *Adaptive K-Means* algorithm.

The first one attempts to force all neurons of the network to participate in the clustering process. This is done by employing a different distance function, which is the one shown in eq. 10. The original squared Euclidean distance is shown in eq. 7.

$$d_{bias}(\vec{x}, \vec{c}^{(i)}) = v^{(i)} * d(\vec{x} - \vec{c}^{(i)}) = v^{(i)} * ||\vec{x} - \vec{c}^{(i)}||^2 \tag{10}$$

The i-th cluster variance, $v^{(i)}$, used to weigh the Euclidean distance, is calculated as in eq. 11. The coefficient α is used much like the learning rate of the network, and usually has a value close to 1, e.g. $\alpha = 0.9999$.

$$v^{(i)}[T+1] = \alpha * v^{(i)}[T] + (1-\alpha) * M^{(i)}(\vec{x}[T]) * ||\vec{x}[T] - \vec{c}^{(i)}[T]||^2 \tag{11}$$

By weighing the Euclidean distance with the variance of each cluster, they manage to bias it in favor of neurons that have responded to fewer data. This is so, because as the number of data won over by a neuron increases, its variance will usually be increasing as well. Therefore, there will come a time, when a neuron with fewer data, and thus a smaller variance, will start winning the others.

As in the case when changes are made to the distance function or the update rule, this new distance function causes a change in the cost function minimized by the network. The new cost function is shown in eq. 12. However, the authors prove that this new *Variance Weighted Mean Square Error*, or *VWMSE* for short, function is equivalent to the *MSE* cost function (see eq. 8). Therefore, the new network has the advantage of better utilizing its neurons, while still retaining the original cost function.

$$\text{VWMSE}(K) = \sum_{i=1}^{K} v^{(i)} * \frac{\sum_{j=1}^{S^{(i)}} ||\vec{x}_j^{(i)} - \vec{c}^{(i)}||^2}{S^{(i)}} = \sum_{i=1}^{K} {v^{(i)}}^2 \tag{12}$$

The second change proposed, has to do with the selection of the *learning rate* and its subsequent adjustment. Since the optimal solution is reached when all cluster variances are

equal, they utilize the *entropy* function, $H(\alpha^{(1)}, \ldots, \alpha^{(K)}) = \sum_1^K -\alpha^{(i)} * \ln(\alpha^{(i)})$, in order to measure the quality of the current clustering. This is because, the *entropy* function takes its maximal value, which is $\frac{1}{K}$, when all its arguments are equal. So, after normalizing the cluster variances, so that they lie in the [0,1] interval (see eq. 13), they define the learning rate, η, of the network as in eq. 14.

$$v_{\text{norm}}^{(i)} = \frac{v^{(i)}}{\sum_{j=1}^K v^{(j)}} \tag{13}$$

$$\eta = \frac{\ln K - H(v_{\text{norm}}^{(1)}, v_{\text{norm}}^{(2)}, \ldots, v_{\text{norm}}^{(K)})}{\ln K} \tag{14}$$

5.4 On the Fly Workload Clustering in CLUE

In addition to the *Optimal Adaptive K-Means* algorithm, *CLUE* provides a version of the more "standard" *Adaptive K-Means* one. The only difference of this version and the one described in subsection 5.1 is that the learning rate is adjusted dynamically, in the same manner as in the *Optimal Adaptive K-Means* algorithm. This *on the fly* algorithm was implemented, in order to compare its performance against the *Optimal Adaptive K-Means*.

A very important characteristic of these neural networks is that they do not need to be trained with the usual time-consuming paradigm of iterating over a set of training data for a long period of time. Instead, they can utilize the clusters, produced by the batch-mode algorithms for the afore-mentioned training data set, to initialize their synaptic weights to the representatives of each cluster, in constant time. This way, the training of the networks can be done very fast and provides very good starting points for the synaptic weights of the neurons.

6 Evaluation of Clustering Results

This section presents the results of the experiments that were performed to evaluate and compare the quality of the clusterings produced by the different algorithms, both on synthetic and real-life traces.

In order to guarantee that the clustering algorithms have been implemented correctly, and behave in an intuitive manner, it is very important to verify that, when their input consists of simple artificial examples, the classes produced are the same with those a human would have identified. If an algorithm does not behave well in these simple cases, it is meaningless to use it with real-life data, since then, the obtained results will probably not be valid. Therefore, a *Test Suite Generator* (*TSG* for short) was built, which constructs sets of artificial traces in the format used by *CLUE*. The input to *TSG* is a specification file, whereas its output serves as input for *CLUE*. The *TSG* specification file is written in a high-level language, and contains information needed to produce a number of triplets, i.e. the areas in the reference matrix that are to be filled, as well as, the particular policy to be followed when filling them (either the whole area is filled or only a random portion of it, and either constant values are used or random ones selected from a user specified interval). The output of *TSG* contains the description of a reference matrix similar to that of fig. 1.

Out of the various metrics [1, 5, 15] used to compare two different clusterings of the same set of data, the most common one is the *square error* criterion, e_i^2. If N data patterns have been partitioned in K clusters $\{C_1, C_2, \ldots, C_K\}$, the square error for the entire clustering is

the sum of the square errors of all clusters (eq. 15):

$$E_K^2 = \sum_{i=1}^{K} e_i^2 \tag{15}$$

6.1 Experiments with synthetic traces

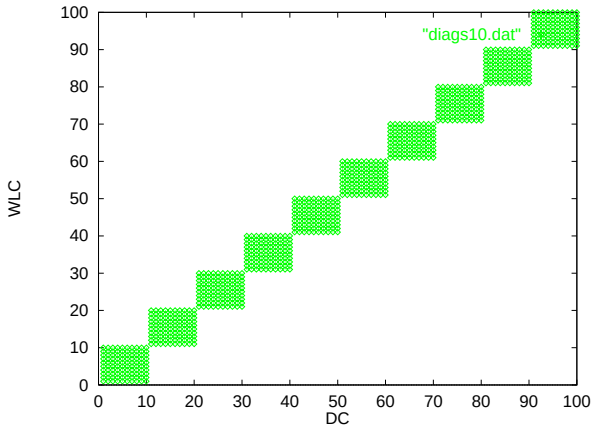

Figure 3: The synthetic trace "diagonal squares".

Various synthetic traces, where the number & composition of workload classes are easy to identify, have been produced using *TSG*. These synthetic traces were subsequently used to compare the various clustering algorithms. Fig. 3 shows the reference matrix for a synthetic trace called "diagonal squares" (since the non-zero entries in the reference matrix form squares along the diagonal of the matrix), where the data classes (DCs) are in the x axis and the workload classes (WLCs) are in the y axis. In this trace, it is evident that the workload classes should be partitioned in 10 clusters, each one containing workload classes from a different square.

Table 1 shows how *HALC*, *K-Means* and *BEA* performed on this type of traces, for different numbers of data (WLCs) and inherent clusters, i.e. different number of squares.

All three algorithms found the expected number of clusters, and both *HALC* and *K-Means* seemed to perform well as the size of the problem gets larger, even though *HALC* outperforms *K-Means*. On the contrary, *BEA* had a long execution time, which renders it unsuitable for large traces.

6.2 Experiments with real-life traces

Except for synthetic traces, we also used two real-life traces, to evaluate the clustering algorithms; in the following, these will be referred to as *DOA* and *PULS*, and were provided by Siemens Nixdorf Informationssysteme AG Germany. These traces where collected during the operation of a simulated single-node relational database system. Table 2 summarizes some of their characteristics.

We assumed that triplets arrive with the frequency measured from the trace, because the traces contain no timing frequency. Measurements of the data locality as seen on the traces showed that a very small number of pages is heavily referenced.

WLC	DC	# clusters	*HALC*	*K-Means*	*BEA*
10	10	1	0.17	0.18	0.21
50	50	5	0.42	0.47	1.99
100	100	10	0.70	0.71	11.18
1000	1000	100	7.49	22.02	8258

Table 1: Comparative results for the three clustering algorithms for the synthetic trace "diagonal squares"

The first column shows how many triplets were in the trace, while the second one, the number of pages that were in the data base (i.e. the dimension of the triplets page reference vector). The third column shows the number of real clusters in the trace, and the other columns show the execution times in seconds for the respective algorithm.

Characteristic	*DOA*	*PULS*
transaction IDs	103	53
user IDs	474	103
terminal IDs	474	110
number of triplets	1984	280
Total of DB pages referenced	41003	66846
non zero reference matrix elements	180017	178749
ref. matrix density	0.22123%	0.95501%

Table 2: Traces *DOA* and *PULS*.

- Observations about locality of references for *DOA*:
  - 10 most referenced pages (0.0243% of all pages) get 16.76% of all references.
  - 80% of all references are to 13.14% of all pages.
  - 90% of all references are to 25.78% of all pages.
- Observations about locality of references for *PULS*:
  - 10 most referenced pages (0.015% of all pages) get 10.02% of all references.
  - 80% of all references are to 26.96% of all pages.
  - 90% of all references are to 45.05% of all pages.

Since the traces were collected on a single node system, in a multiple node TP system special handling of the pages was needed. Therefore we assigned pages to a number of buckets (disk-resident files) so that, according to the access info on the trace, each bucket is accessed with the same frequency.

The four clustering algorithms, namely *RANDOM*, *HALC*, *VWMSE* (Optimal - Adaptive K-Means) and *K-Means* [4], were used to produce a maximum number of 30 clusters. Figures

[4] *BEA* was excluded, since its execution time was too high for it to be of any practical use.

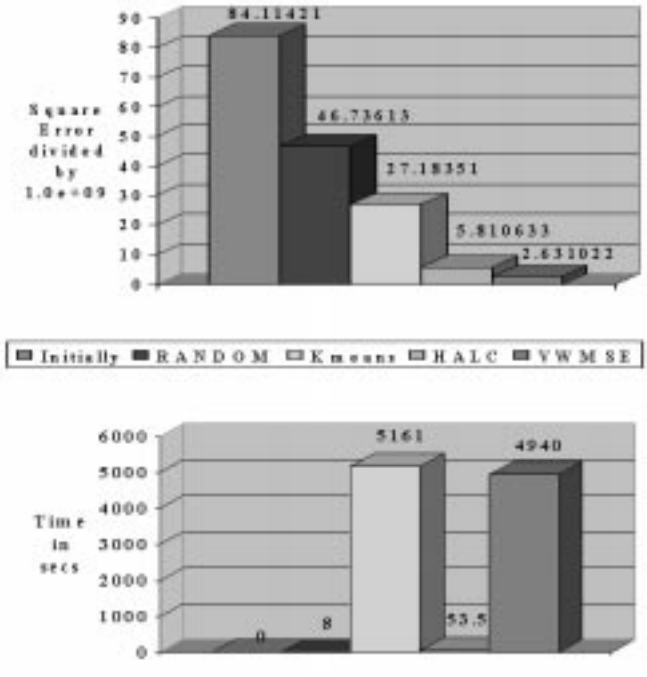

Figure 4: Comparative results for the four clustering algorithms using the *DOA* trace.

4 and 5 show how the four clustering algorithms performed in terms of execution time (measured in seconds) and the square error criterion.

For the *VWMSE* clustering algorithm, which is based on neural networks, first both the traces (*DOA* and *PULS*) were divided in two different sets. The *training set* of a trace was used for the training of the neural network, and the *testing set* for the evaluation of the clustering. For the *DOA* trace 10% of the original trace triplets comprised the training set and the other 90% the testing set. For the *PULS* trace the training set was exactly 30 triplets (in order to initialize all the neurons) and the remaining triplets were the testing set.

As Figures 4 and 5 suggest, *HALC*, *VWMSE* and *K-Means* managed to construct useful clusters (i.e. with a much smaller square error than that of the *RANDOM* algorithm), even though *HALC* and *VWMSE* seem to produce somewhat better solutions. Additionally, *HALC* takes a lot less time to finish than *K-Means* and *VWMSE*. Execution times for *HALC* are in the order of seconds, while for *K-Means* and *VWMSE* are in the order of minutes. *K-Means* produces in both cases a very big cluster containing most of the pages, caused from the fact that in both traces triplets are accessing the same pages with high probability.

7 Impact of Workload Clustering on Transaction Routing

This section provides a detailed description of the simulation model that was used, and the experiments performed in order to investigate the impact of the quality of clustering on the performance of various transaction routing policies.

7.1 Simulation Methodology

The evaluation study is based on the *TPsim* [19] simulator of multiple-node shared-nothing transaction processing systems. The simulator was written in C, on top of a threads-based

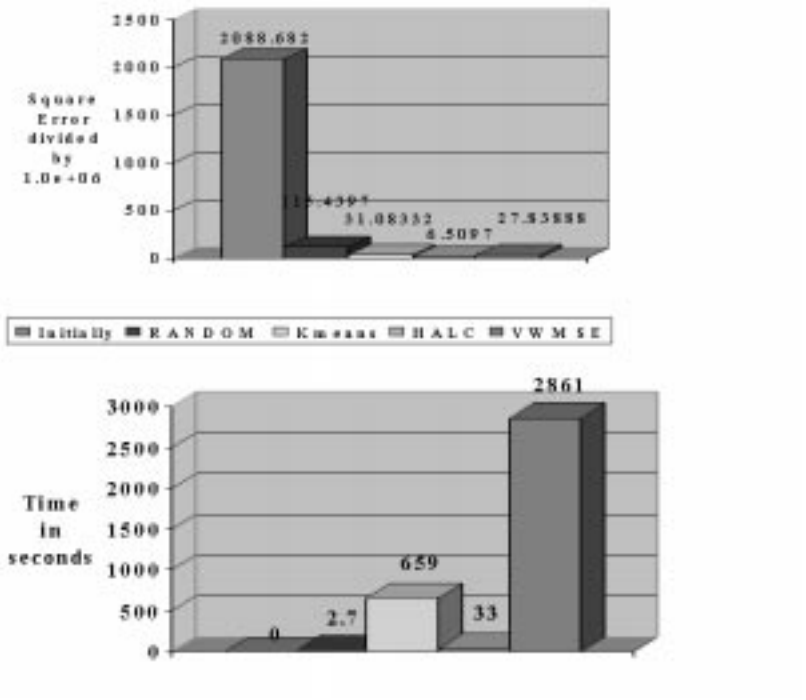

Figure 5: Comparative results for the four clustering algorithms using the *PULS* trace.

simulation support library (*PARASOL* [23]), and a parser that processes a high-level description of the system configuration and workload, in order to configure the simulated run-time environment according to user specifications. *TPsim* models a transaction processing monitor coupled with a database manager, and fully emulates concurrency control (two-phase locking with deadlock detection), LRU buffer management, logging (including group commit), CPU scheduling, I/O scheduling and distributed two-phase commit. It allows simulation of multiple-class workloads and collects performance related statistics for each workload class.

Initial experiments showed that since there were some transactions accessing thousands of pages and the rest only accessed a few pages, the results were strongly affected from those long-lived transactions. Since these transactions needed to lock a big part of the database, many others were blocked waiting for locks owned by these transactions. This caused a lot of timeouts and deadlocks between the transactions and as a result low utilizations. So we reduced the traces by removing instances of triplets that were accessing more than 30 different pages. For the remaining triplets, the overall number of page references per transaction can be much higher than 30, however each transaction references no more than 30 different pages.

The resulting traces, which will be called *Reduced DOA* and *Reduced PULS* from now on, consist of 1836 triplets (92.5% of the initial trace *DOA*) and 144 triplets (51.4% of the initial trace *PULS*), respectively. The average number of page references per transaction is 27 for the *Reduced DOA* trace, and 55 for the *Reduced PULS* trace.

The OLTP system configuration used in the experiments consisted of 7 nodes for the *Reduced DOA* trace, and of 5 nodes for the *Reduced PULS* trace. Incoming requests from users were routed by a dedicated "front-end" node to one of the other 6 or 4, respectively, available "back-end" nodes, according to the policy adopted for transaction routing. There were 100 user terminals in the system, submitting streams of units of work for service. Each user terminal was modeled as a source that submits a request, waits for the response, and then waits for a period of time (*think time*) before submitting the next request. This means

that the maximum number of active transactions was limited only by the multiprogramming level. All the nodes of the simulated system were identical although some experiments were made with heterogeneous clusters. The main configuration parameters, with their settings for each of the nodes, as well as the ones for the connecting network, are shown in Table 3.

We varied CPU speed from 20 - 28 MIPS in order to get results for different (high) utilizations. Because of the locality of references issued by the transactions in both *Reduced DOA* and *Reduced PULS*, pages were found with a high probability in the private buffers of the node and as a result disk utilizations were very low (usually lower than 15%). The communication network also was fast enough so as not to become a bottleneck. All the results were taken only after the end of the first 10000 transactions, so that the measurements taken would correspond to a stable system state. The simulation time was chosen to be 2000 time units. During this period, the simulated system completed about 150.000 transactions of *Reduced DOA* and 80.000 of *Reduced PULS* (in *Reduced PULS* transactions make almost twice as many page references as in *Reduced DOA*). Due to lack of knowledge about the composition of each transaction class (which depended also by the clustering algorithm used), we chose the same performance goal for all transaction classes. The class performance goal parameters are taken into account by the goal-oriented routing algorithms [10].

Reduced DOA	
number of CPUs	1
MPL	400
number of disks	1
CPU speed	20 - 24 - 28
Private buffer	30% of local pages
Communication Network:	
packet size	1024 bytes
transfer rate	80Mbits
Workload	
Transaction Classes	30
Tx Appl. Burst	500000.0
Users	100
Think time	0.014
Performance Goal	0.5

Reduced PULS	
number of CPUs	1
MPL	400
number of disks	1
CPU speed	20 - 25 - 28
Private buffer	50% of local pages
Communication Network:	
packet size	1024 bytes
transfer rate	80Mbits
Workload	
Transaction Classes	30
Tx Appl.Burst	1000000.0
Users	100
Think time	0.038
Performance Goal	1.0

Table 3: Main simulation parameter settings.

The objective of the experiments was to get an evaluation of the simulated system's performance, with respect to alternative workload clusterings and transaction routing algorithms. Each incoming request carried a workload class identifier that was assigned by a workload characterization module. The latter provided a profile of anticipated resource demands for each class, as well. Workload classes were determined from the clustered triplets, either of *Reduced DOA* or *Reduced PULS*.

Afterwards, *CLUE* was used to cluster the triplets. This information was subsequently used by the workload generation module of the simulator, which produced streams of triplets, where each triplet's reference pattern was (on average) as computed from the original trace, and its frequency of occurrence as derived from the original trace. The transaction router,

however, had to base its decision on the expected resource consumption profile associated with the (statically assigned) workload class identifier for the triplet, which was taken from the output of *CLUE*. Thus, the quality of clustering, by affecting what the router perceived as the expected resource demands for each incoming unit of work, could have a marked impact on system performance.

7.2 Transaction Routing Algorithms

The transaction routing algorithms used in the experiments were *WFW*, *WFWC*, *SGOR*, and *JSQ*, which are described in detail in [10]. These are all dynamic control algorithms, able to handle multiple-class workloads. Both *WFW* and *WFWC* aim at minimizing the response time for each arriving transaction. *WFW* is based on the Join-the-Shortest-Queue (*JSQ*) algorithm but takes into account data affinity, as well. *WFWC* is based on *WFW* but also considers the priority of each arriving transaction, as well as, of those already in the system. *SGOR* tries to minimize the maximum performance index (see subsection 7.3) over all classes, by estimating the effect of possible routing decisions on the performance index of all classes.

All the above algorithms (except *JSQ*) use an estimation of the transaction response time, which is the sum of the estimation of the service time, determined by an expected resource consumption profile and the estimation of the queueing delay, derived from current run-time state information. The resource consumption profile, used for the service time estimation, is associated with the workload class to which the unit of work to be routed has been assigned. Such a profile includes the following information:

- the average CPU work $\mathrm{W}(i,l,k)$ generated on node S_k by a transaction of class C_i, which was routed to node S_l,
- the average number of times $\mathrm{V}(i,l,k)$ a transaction of class C_i, which was routed to node S_l, "visits" node S_k,
- the expected total I/O delay $\mathrm{I}(i)$ of a transaction of class C_i,
- the expected total communication delay $\mathrm{D}(i,l)$ of a transaction of class C_i, which was routed to node S_l,
- the expected total I/O delay $\mathrm{C}(i,l)$, due to writing log records related to the commit protocol, at all sites for a transaction of class C_i, which was routed to S_l.

However, the values for these parameters were not available for the traces described in subsection 6.2. Therefore, in order to obtain meaningful values for the above parameters, a series of *monitoring simulation runs* were made first, one for each clustering algorithm and each trace file. The scheduling algorithm used in these monitoring runs was round-robin, and the routing of transactions was done randomly, while the rest of the settings were the same as for the normal simulation runs. This way, it was possible to obtain measurements of the above quantities, which were subsequently used as an approximation of the resource consumption profiles required by the routing algorithms. This approach was deemed necessary, as the original system from which the traces were collected is no longer available to the authors.

An important notice on the performance of these algorithms is that goal-oriented algorithms that attempt to minimize the maximum performance index, tend to cause the minimum and maximum performance index curves to converge at high throughput. This causes the increase of the throughput at which goals can be met, compared to non-goal oriented algorithms (such as WFW), but also results in an increased average transaction response time over all transaction classes (the metric we chose to measure).

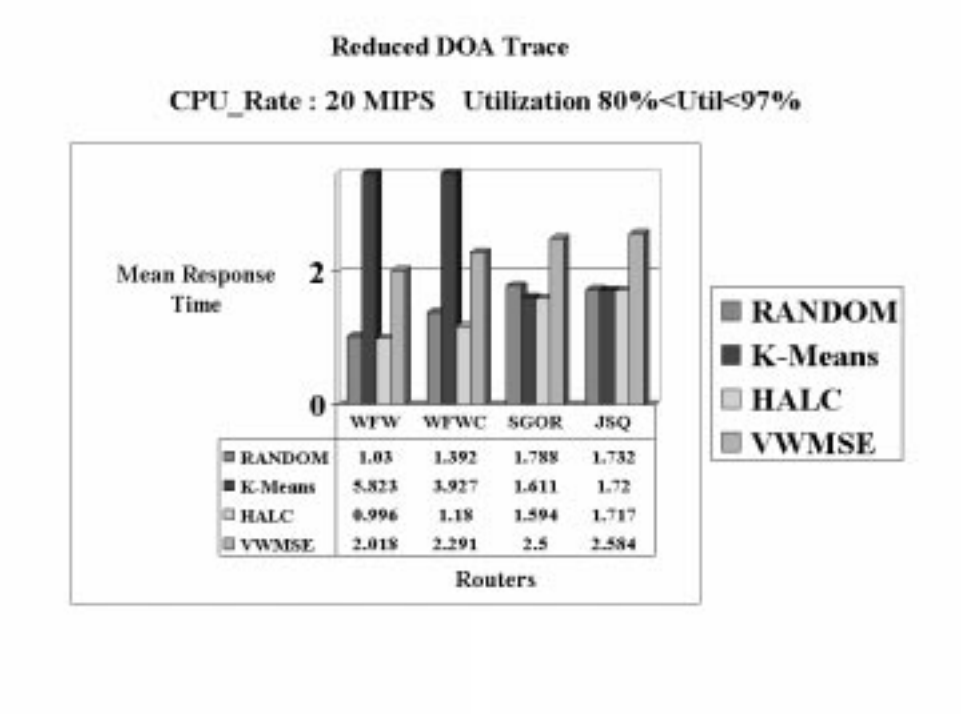

Figure 6: Mean response time over all the transaction classes for the *Reduced DOA* trace.

7.3 Metrics and Experimental Results

The main metric that we considered in this evaluation was the average response time over all the units of work.

The performance index P_i of a transaction of class C_i, for which a performance goal g_i has been defined (maximum acceptable response time for transactions of class C_i), is defined [10] as in eq. 16:

$$P_i = \frac{RT_i}{g_i} \tag{16}$$

In eq. 16, RT_i is the current estimation of the average response time for transactions of class C_i. RT_i gets updated, whenever a transaction T of class C_i is terminated, as shown in eq. 17:

$$RT_i \leftarrow (1-\alpha) \cdot RT_i + \alpha \cdot R(T) \tag{17}$$

In eq. 17, $R(T)$ is the response time of T and α is a constant ($0 < \alpha \leq 1$) that weighs the relative importance of recent measurements of response time, against measurements of response times for units of work that completed further back in time. In these experiments, α was set to be 0.8.

The CPU scheduling algorithm used in these experiments was CLASS_PI [20]. This CPU scheduling algorithm sets the priority of a transaction (instance of a particular class) equal to $\frac{1}{P_i}$, where P_i is the current performance index of the class. Therefore the CPU scheduling algorithm is an adaptive policy, as it considers satisfaction of class performance goals. It favors transactions belonging to classes that at a particular decision instant, are more probable to exceed their specified response time goal.

7.3.1 Experimental results for the *Reduced DOA* trace

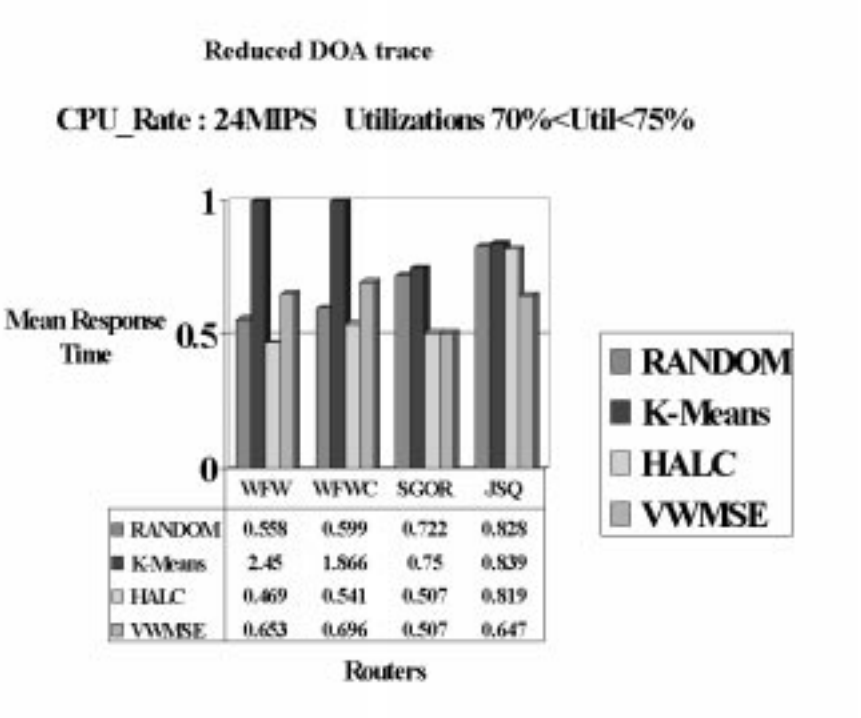

Figure 7: Mean response time over all the transaction classes for the *Reduced DOA* trace.

The experimental results for the *Reduced DOA* trace can be seen in the three charts presented in Figures 6, 7 and 8. The metric in these charts is the mean response time over all transactions. The CPU rate of the processors for the nodes of the simulated system was for the first chart 20 MIPS (Million Instructions Per Second), for the second 24 MIPS and for third one, 28 MIPS. In this way we measured the performance of the system under different utilizations.

The performance of the Join-Shortest-Queue (JSQ) router provides the basis of comparison, because the router does not take into account the affinity of a transaction class to a given database partition. According to this policy, a newly arrived transaction is routed to the node with the minimum number of active transactions.

The experiments which used the *RANDOM*, *K-Means* and *HALC* clustering algorithms with the JSQ router, resulted in mean response times that are very close. This was to be expected, because the JSQ router does not take into account the information produced by clustering. Although someone would expected that the node utilizations, using the JSQ router, would be about the same for all the nodes, experiments showed that a node (in particular node_3) had a much lower CPU utilization than the others. This happened because many transactions running on the other nodes were making requests to this node ("function-shipping" [30]). Function-shipping to a node is made in order to access non-local data in a Shared-Nothing transaction processing system. It results in creating a mirror transaction on that node which imposes less overhead than the (primary) transaction. The primary transaction is located on the node that the router chose to service the transaction and also controls the two-phase commit protocol.

For the *VWMSE* clustering algorithm, the testing set of the *Reduced DOA* trace (90% of the triplets of *Reduced DOA*) was used for collecting measurements of transaction response times. As mentioned in the previous section, triplets in the testing set issued on average 5 DBcalls per transaction more than the triplets in *Reduced DOA* trace. Using the JSQ

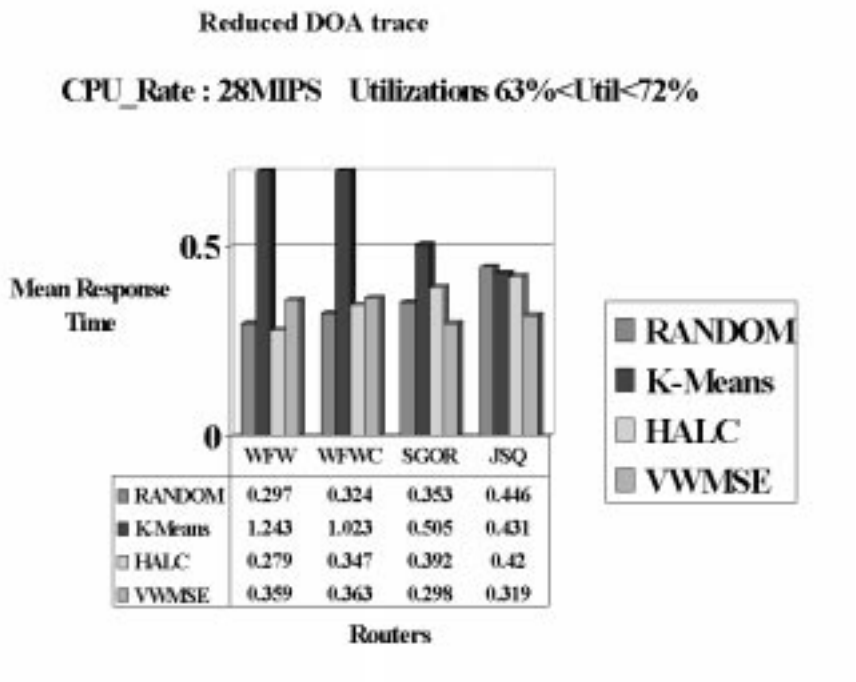

Figure 8: Mean response time over all the transaction classes for the *Reduced DOA* trace.

router for the 20 MIPS case, the node utilizations were around 97% while for the other clustering algorithms the node utilizations were about 90%. This caused a higher probability of deadlocks among the transactions and, as a consequence, much worse overall mean response time. As the utilization of the nodes decreased (see Figures 7, 8), the performance of the system reached and finally exceeded the performance of the systems that used the *RANDOM*, *K-Means* and *HALC* clustering algorithms. The reason for this behavior is that using the testing set, nodes had more balanced utilizations[5] and as a result better mean response time. The same behavior was also observed with the SGOR router which, while trying to minimize the maximum performance index, did not take into account load imbalance.

The JSQ router showed the worst performance among the four routers used in these experiments. Only in the experiments which used the *K-Means* clustering algorithm, JSQ showed a better performance. This was because *K-Means* resulted in very bad clusters (95% of the triplets were assigned in one cluster). The bad quality of clustering affected the performance of the system dramatically, resulting in very high overall mean response time. Only in the experiments with the SGOR router, the system managed to handle the workload, resulting in acceptable mean response time.

The WFW router resulted in the lowest overall mean response time observed, for the clustering algorithms used, except for the case of *K-Means*. In general WFWC came next, followed by SGOR and JSQ. From the different clustering algorithms that were used in these experiments, *HALC* proved to be the best when the router considered data affinity.

The results show that clustering the triplets in a random manner had better results than clustering them using *K-Means* or *VWMSE* clustering algorithms. We must keep in mind that *RANDOM* clustering benefits from the fact that triplets in the trace have widely varying probabilities of appearance. The assignment of a frequently appearing triplet in a class with

[5]Node_3 had a much higher utilization because part of the transactions accessing data stored on this node were not part of the testing set.

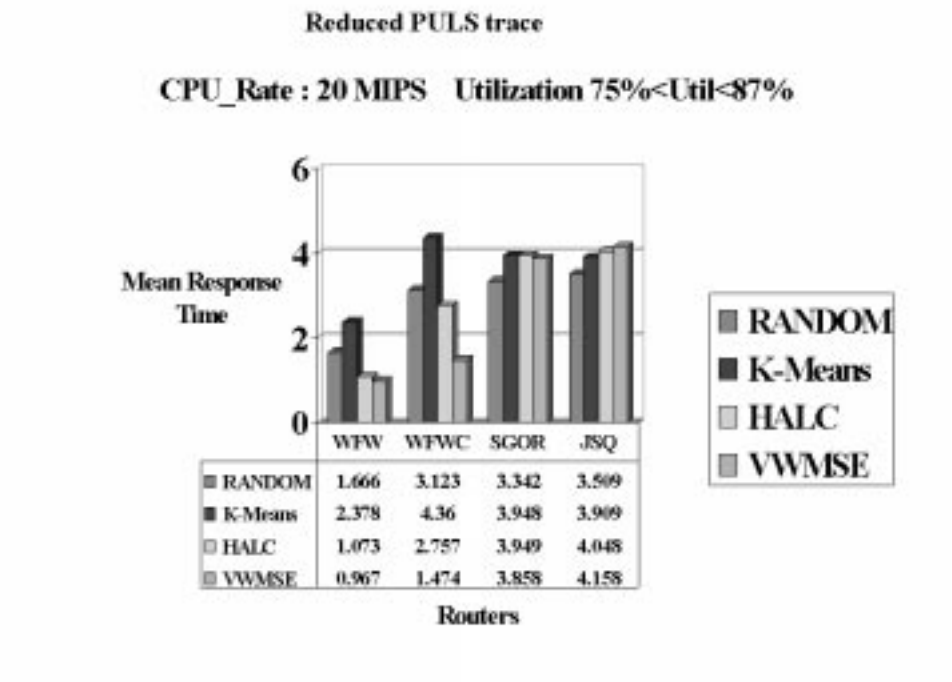

Figure 9: Mean response time over all the transaction classes for the *Reduced PULS* trace.

other triplets that have much lower probabilities of appearance causes the class measurements taken in the monitoring run to be dominated from the affinity of these frequently appearing triplets. The routers (except JSQ) use these measurements in order to route an instant of a transaction class, to the node to which the transaction class had a higher affinity. As a result, although we use a *RANDOM* clustering algorithm, many of the triplets with high probability of appearance were routed to the their "favorite" (in terms of data affinity) node. This has a considerable impact on the performance of the system, as the number of function-shipped requests is much lower.

7.3.2 Experimental results for the *Reduced PULS* trace

More or less the same observations hold also for the performance of the simulated systems using the *Reduced PULS* trace. The experiment results for the *Reduced PULS* trace can be seen in the three charts presented in Figures 9, 10, 11.

Here the number of classes (30) is much closer to the total number of triplets (144) and so the classes were consisted of a much smaller number of triplets (for *RANDOM* clustering algorithm the number of triplets was 4-5 per class). Even in this case, *HALC* clustering algorithm is the clear winner among the clustering algorithms used. The simulation runs made using the *K-Means* clustering algorithm to cluster the trace data, suffered again from the bad quality of clustering although the mean response time is much closer to those measured using the other clustering policies. This is caused from the fact that in the *Reduced PULS* trace the $\frac{num.of classes}{num.of triplets}$ ratio is much bigger compared to that of the *Reduced DOA* trace, and so K-means had at least $\frac{29}{144} \cdot 100$ triplets in different classes (about 20%). Mean response time over all classes is also much higher as a result of the higher average number of database calls issued from the transactions consisting the *Reduced PULS* trace.

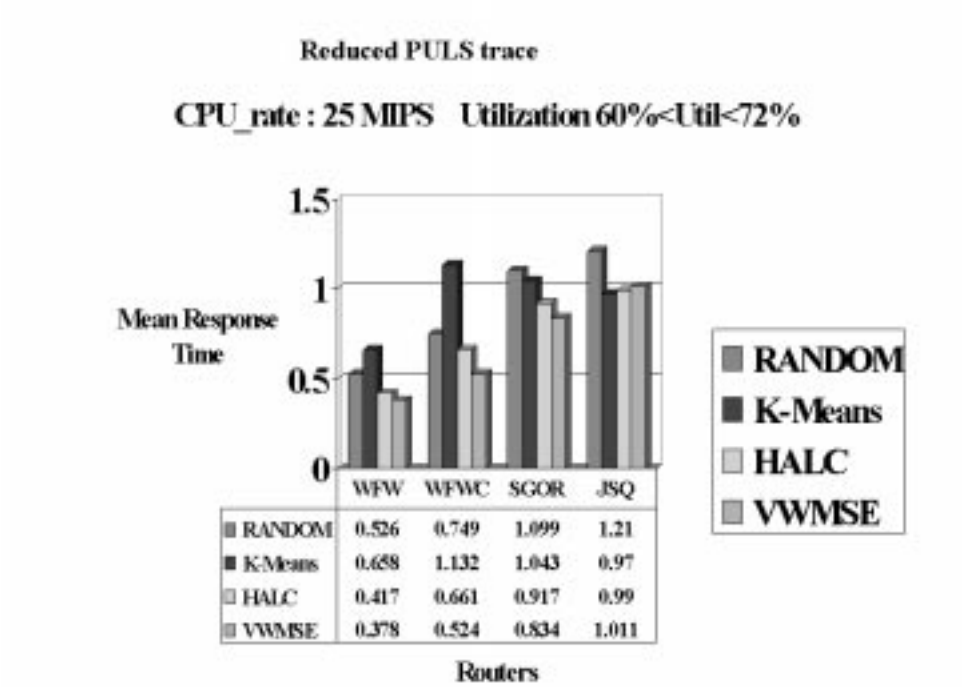

Figure 10: Mean response time over all the transaction classes for the *Reduced PULS* trace.

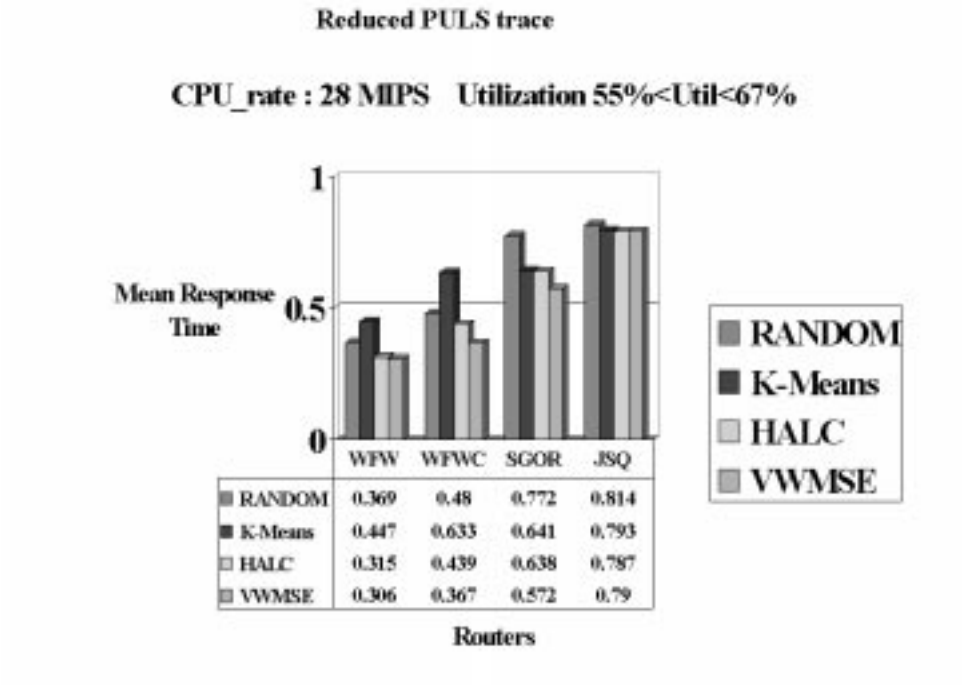

Figure 11: Mean response time over all the transaction classes for the *Reduced PULS* trace.

8 Conclusions

In large-scale transaction processing systems, knowledge of the workload intrinsic characteristics is essential for performance management purposes. In these systems, dynamic workload control algorithms are used, in order to optimize performance. These algorithms can benefit from the results of workload clustering algorithms that partition the workload into classes exhibiting similar characteristics.

Four different routing algorithms along with four clustering algorithms were considered in this paper. For this evaluation, two traces from real-life OLTP systems were used. Experiments showed that the heuristic clustering algorithm (HALC) and the Optimal Adaptive K-Means (which is based on neural networks) performed very well in terms of quality of clustering. On the other hand, the classic K-Means algorithm produced disappointing results.

The impact of the above workload clusterings on the performance of three dynamic transaction routing algorithms (plus a Join-Shortest-Queue router for comparison reasons) was also studied. The quality of workload clustering reflected the performance of the routing algorithms. HALC clustering algorithm improved a lot the performance of these algorithms. Simulations with K-Means showed how a bad clustering can degrade the performance of systems that use dynamic goal-oriented routing algorithms. Optimal Adaptive K-Means did not perform very well, but this was probably because of the experiment settings and in particular from the fact that the simulated system used a subset of the trace, called testing set. We have reasons to believe that Optimal Adaptive K-Means can perform much better, and our belief is based on the quality of workload clustering that it produces. This will be part of our future work since *on the fly* clustering algorithms appear to be well-suited for OLTP systems with workload characteristics that change with time.

Acknowledgments

The authors would like to express their gratitude to Thomas Delica from Siemens Nixdorf Informationssysteme, for providing the traces that were used in the afore-mentioned simulations, and to Maria Karavassili and Anastasia Anastasiadi, former members of the PLEIADES research group at ICS/FORTH, who helped link the two tools (CLUE and TPsim) and collect the simulation results. The work presented in this paper was supported by the European Union ESPRIT III Basic Research Project LYDIA 8144.

References

[1] M. R. Anderberg. *"Cluster Analysis for Applications"*. Academic Press Inc., New York, 1973.

[2] Eike Born, Thomas Delica, Werner Ehrl, Lutz Richter, and Reinhard Riedl. "Characterization of Workloads for Distributed Processing - Methodology and Guide". Deliverable WP2/T.2.1/D4 of project LYDIA (available online at the URL: `http://www.ics.forth.gr/proj/pleiades/projects/LYDIA/1st-year_deliv/4thDel.ps`), June 1995.

[3] M. Calzarossa and L. Massari. "Measurement-Based Approach to Workload Characterization". In R. Marie, G. Haring, and G. Kotsis, editors, *Tutorial Papers of the 7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, pages 123–147, Vienna, 1994. Springer-Verlag.

[4] M. Calzarossa and G. Serazzi. "Workload Characterization: A Survey". *Proceedings of the IEEE*, 81(8), August 1993.

[5] S. Chakravarthy, J. Muthutaj, R. Varadarajan, and S. Navathe. "An Objective Function for Vertically Partitioning Relations in Distributed Databases and its Analysis.". Technical Report UF-CIS-TR-92-045, University of Florida, Computer and Information Sciences, 1992.

[6] Chedsada Chinrungrueng and Carlo H. Séquin. Optimal Adaptive K-Means Algorithm with Dynamic Adjustment of Learning Rate. *IEEE Transactions on Neural Networks*, 6(1):157 – 169, January 1995.

[7] Christian J. Darken and John Moody. Fast adaptive k-means clustering: Some empirical results. In *Proc. International Joint Conference on Neural Networks (IJCNN-90)*, June 1990.

[8] Christian J. Darken and John Moody. Learning schedules for stochastics optimization. In *1990 IEEE Conf. Neural Information Processing Systems-Natural and Synthetic*, November 1990.

[9] D. Desieno. Adding a conscience to competitive learning. In *Proc. 2nd IEEE International Conference on Neural Networks (ICNN-88)*, volume I, July 1988.

[10] D. Ferguson, C. Nikolaou, L. Georgiadis, and K. Davies. "Satisfying Response Time Goals in Transaction Processing Systems". In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, 1993.

[11] D. Ferrari, G. Serazzi, and A. Zeigner. *"Measurement and Tuning of Computer Systems"*. Prentice-Hall, New Jersey, 1983.

[12] Allen Gersho. Asymptotically optimal block quantization. *IEEE Transactions on Information Theory*, IT-25(4):373–380, 1979.

[13] John A. Hartigan. *Clustering Algorithms*. John Wiley and Sons, Inc., 1975.

[14] W. Y. Huang and R. P. Lippman. Neural net and traditional classifiers. In D. Z. Anderson, editor, *Neural Information Processing Systems*, pages 387–396. American Institute of Physics, 1988.

[15] A. Jain and R. Dubes. *"Algorithms for Clustering Data"*. Prentice-Hall, New Jersey, 1988.

[16] O. Klaassen. "Modeling Database Reference Behaviour". In G. Balbo and G. Serrazi, editors, *Computer Perfomance Evaluation: Modeling Techniques and Tools*, pages 47–60. North-Holland, 1992.

[17] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, IT-28(2):129–137, March 1982.

[18] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. 5th Berkeley Symposium on Mathematics, Statistics, and Probability*, volume 1, pages 281–298, 1967.

[19] Manolis Marazakis. Simulation of transaction processing systems and a study of methods for performance goal satisfaction. Master's thesis, Dept. of Comp. Science, Univ. of Crete, Greece, P.O. 1470, Heraklion, Crete, Greece, October 1995.
Also: Technical Report No. TR95-0140, ICS-FORTH, Heraklion, Crete, Greece.
(Both in Greek)
Note: ICS Technical Reports can be obtained through the ICS/Pleiades `Dienst` server at the URL: http://www.ics.forth.gr/TR.

[20] Manolis Marazakis and Christos Nikolaou. Towards adaptive scheduling of tasks in transactional workflows. Technical Report No. 134, ICS-FORTH, Heraklio, Crete, Greece, August 1995.

[21] W. McCormick, P. Schweitzer, and T. White. Problem Decomposition and Data Reorganization by a Clustering Technique. *Operation Research*, 20, September/October 1972.

[22] John Moody and Christian J. Darken. Fast Learning in Network of Locally-Tuned Processing Units. *Neural Computation*, 1(2):281–294, 1989.

[23] J. E. Neilson. "PARASOL: A Simulator for Distributed and/or Parallel Systems.". Technical Report SCS-TR-192, Carleton University, Canada, 1991.

[24] K. E. E. Raatikainen. "Cluster Analysis and Workload Classification". *Performance Evaluation Review*, 20(4):24–30, May 1993.

[25] E. Rahm. "A Framework for Workload Allocation in Distributed Transaction Processing Systems". *J. Systems Software*, 18:171–190, 1992.

[26] David E. Rumelhart and James L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume Foundations. MIT Press, 1986.

[27] David E. Rumelhart and David Zipser. *Feature Discovery by Competitive Learning*, chapter 5, pages 151–193. Volume Foundations of Parallel Distributed Processing [26], 1986.

[28] N. B. Venkateswarlu and P. S. V. S. K. Raju. Fast ISODATA Clustering Algorithms. *Pattern Recognition*, 25(3):335–342, 1992.

[29] P. S. Yu, M. S. Chen, H. U. Heiss, and S. Lee. "On Workload Characterization of Relational Database Environments". *IEEE Transactions on Software Engineering*, 18(4):347–355, April 1992.

[30] P. S. Yu and A. Dan. "Impact of Workload Partitionability on the Performance Coupling Architectures for Transaction Processing". In *Proc. of the 4th IEEE Int. Symp. on Parallel and Distributed Processing*, pages 40–49, Arlington, Texas, December 1992. IEEE Computer Society Press.

[31] P. S. Yu and A. Dan. "Performance Analysis of Affinity Clustering on Transaction Processing Coupling Architecture". *IEEE Transactions on Knowledge and Data Engineering*, 6(5):764–786, oct 1994.