Distributed and Parallel Databases, 10, 161–198, 2001

Logically Clustered Architectures for Networked Databases

JE-HO PARK*
VINAY KANITKAR†
ALEX DELIS
Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201, USA

Recommended by: Athman Bouguettaya

Abstract. By effectively harnessing networked computing resources, the two-tier client-server model has been used to support shared data access. In systems based on this approach, the database servers often become performance bottlenecks when the number of concurrent users is large. Client data caching techniques have been proposed in order to ease resource contention at the servers. The key theme of these techniques is the exploitation of user data access locality. In this paper, we propose a three-tiered model that takes advantage of such data access locality to furnish a much more scalable system. Groups of clients that demonstrate similarities in their data access behavior are *logically* clustered together. Each such group of clients is handled by an *Intermediate Cluster Manager* (ICM) that acts as a cluster-wide directory service and cache manager. Clients within the same cluster are now capable of sharing data among themselves without interacting with the server(s). This results in reduced server load and allows the support of a much larger number of clients. Through prototyping and experimentation, we show that the logical clustering of clients, and the introduction of the ICM layer, significantly improve system scalability as well as transaction response times. Logical clusters, consisting of clients with similar data access patterns, are identified with the help of both a greedy algorithm and a genetic algorithm. For the latter, we have developed an encoding scheme and its corresponding operators.

Keywords: logical client clustering, networked databases, multi-tier database architectures

1. Introduction

Database systems in modern networked environments are required to manage tremendous volumes of data and allow transparent location-independent access to it. Applications deployed in such environments include CAD/CAM, computer integrated manufacturing, systems for the management of production, and electronic-commerce systems. Typically, a large number of users work simultaneously in order to complete design new components, track provided services, and oversee financial operations [29, 30, 34, 45]. Efficient database support for these systems is crucial. Previous work in the area has examined the relationship between application software and independent data-servers [5, 17, 39, 51]. Although the study of this interaction is essential, the handling of high-volume data among various sites

*Present address: Voicemate Inc. 95 Morton Street, New York, NY 10014.
†Present address: Akamai Technologies, 500 Technology Square, Cambridge, MA 02139.

in a network-based infrastructure poses new challenges [50]. To meet these challenges, contemporary databases have been based on the client-server (CS) model where a number of machines (servers) host data and others (clients) carry out transaction processing [11, 14].

Client-server databases have shown reduced response times for client transactions over their centralized counterparts. However, as data servers are shared among many users, they become points of contention. Studies in this area have shown that when the number of clients attached per server becomes large, such database architectures fail to guarantee satisfactory performance rates [14]. Transient data caching at clients has been used as a mechanism for improving system response times and easing resource contention at database servers. By storing frequently accessed data at the client sites, repeated requests to the server for the same data can be avoided, if not completely eliminated. The emergence of high speed networks has also created new opportunities in CS systems since the contents of remote client caches can also be exploited as data resources [35]. Data request forwarding schemes that allow client requests to be treated by other clients, instead of servers, have been introduced in distributed file systems and object-oriented client-server databases [6, 13]. Such request forwarding has been shown to improve transaction throughput rates and make use of idle client resources. Although the above efforts have helped improve the performance of CS systems, there are still upper bounds on their scalability. This is because servers are required to not only process client data requests, but also manage global locking and maintain data-location directories. As larger numbers of clients are attached per server, delays experienced by clients in obtaining server data increase considerably and contribute to a marked degradation in throughput.

In this paper, we propose two alternative architectures that avoid this performance degradation by off-loading server tasks to an intermediate layer in the access hierarchy. The key feature of the resulting three-tier configurations is *logical* client clustering. By analyzing data access patterns of involved sites, we can logically group clients into disjoint sets. Clients that access similar segments of the database are grouped together. Each cluster of clients is handled by an *Intermediate Cluster Manager* (ICM) which is connected to the existing database server(s). The interaction among clients and server(s) is achieved with the help of ICMs that may feature a data-cache on their own. An ICM provides the following services to its member clients:

– A cluster-wide data directory is maintained so that requests can be forwarded to other clients within the cluster, whenever possible.
– Concurrency related structures (i.e., lock tables, cluster-wide wait-for graphs) for data objects in a cluster are maintained at the ICM level. These structures are also used to carry out deadlock detection within clusters.
– If an ICM has its own cache, requests for data/locks available in the cache can be handled by the ICM itself.

By generating good logical client clustering and performing the above functions using ICMs, the load on the server can be reduced significantly. This results in improved transaction response times and consequently better system scalability. In general, it is worth mentioning that the problem of logical client clustering is computationally expensive. In fact, the optimal clustering of clients based on data access patterns is known to be NP-Complete [21, 44].

We have implemented system prototypes for the standard client-server and the three-tier database architectures. We have also implemented greedy and genetic algorithms to generate favorable client clusters. Our experimental results establish that the clustered three-tier systems avoid the scalability problem encountered by the conventional CS implementations. Under a number of diverse workloads, the clustered configurations demonstrate many-fold reductions in observed transaction turnaround times. Only in the case where good client clustering was difficult to generate, the standard two-tier CS configuration featured comparable performance.

Client clustering has been considered in the past by the operating system and database communities. Many techniques that advocate the use of clusters of computing nodes and/or TP monitors in order to scalably provide services to a large number of users have been proposed. Such configurations have been used in the implementation of highly available web servers and database servers [1, 16]. Logical clustering, as proposed in this paper, is different from earlier clustering techniques in that it does not advocate the creation of computing clusters based on the physical location or proximity of the involved client sites. We believe that as network availability and bandwidth increases, the importance of clients' physical proximity will continue to diminish. And, as data transfer costs and overheads become lower, managing the sharing of data and co-ordinating serializable database accesses become significant areas for improvement. These are the areas that we seek to address with logical client clustering.

The remainder of this paper is organized as follows. Section 2 describes the key characteristics of the standard CS as well as two alternative ICM-based architectures. In Section 3, we present the algorithms used to generate client clusters and Section 4 discusses our experimental methodology and presents our performance results. Section 5 outlines related work and compares it with our approach. Conclusions can be found in Section 6.

2. Networked database architectures

This section outlines two database architectures built around the concept of the *Intermediate Cluster Manager* (ICM) and contrasts them with the conventional Flat CS architecture (FCS). The two three-tiered configurations are the Logically Clustered CS database (LC-CS) and its extended version (Extended-LC-CS). In the former, ICMs provide directory services to allow clients within clusters to share data while the latter also allows data and lock caching at the ICM level. Objects are the unit of data transfer between the server and the client workstations [11, 15]. For the sake of simplicity, we assume that the granularity of objects remains constant. All three configurations share a number of features:

(i) The primary copy of the database is hosted by the server and the clients communicate with it via IPC (Inter-Process Communication) abstractions. There may be more than one database server but we assume that there is no replication of data among them.
(ii) User transactions are initiated at the clients. The required data are fetched by the clients and used by the transactions locally. Concurrent transaction processing is permitted at each client site. Each client uses its memory and disk space to maintain local copies of objects.

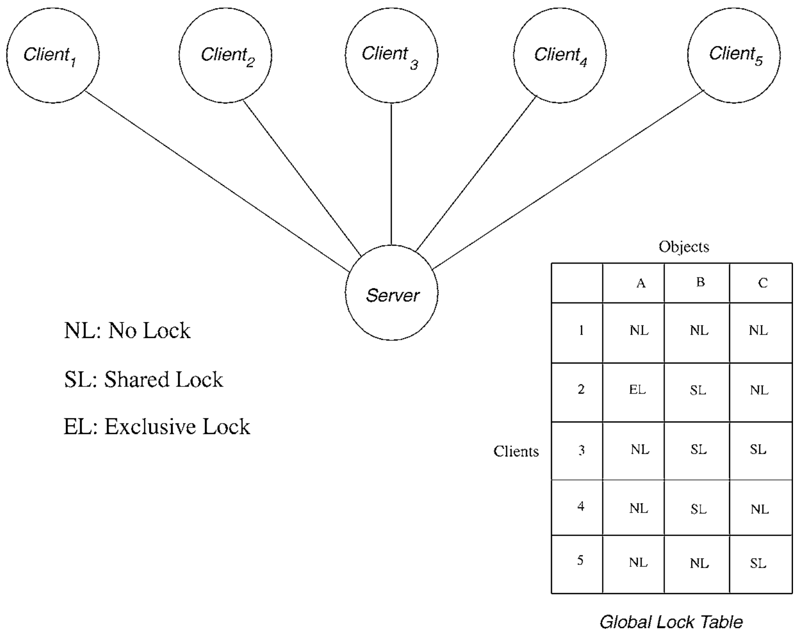
(iii) The server performs low-level database functionalities on behalf of requesting clients. In addition, the server ensures that all accesses to the data are serialized and that there are no operations being performed on stale copies of data objects. This is achieved by maintaining a global lock table.
(iv) Inter-transaction caching at the clients is permitted, i.e., database objects are allowed to remain in client caches across transaction boundaries [14, 40].

In each architecture, two types of locks are allowed: shared (read-only) and exclusive (read-write) [43]. Several transactions can access the same data item with a shared lock (SL). On the other hand, at any given time, only one client is allowed to lock a data object exclusively. Moreover, a transaction is not allowed to obtain any type of lock on an object that has been locked by another client or transaction in an exclusive mode (EL). The locking scheme has been derived from the strict *two phase locking* (2PL) mechanism for distributed environments [4]. The following three subsections describe the operations of the three architectures in detail.

2.1. Flat client-server architecture (FCS)

The flat client-server architecture consists of a number of clients directly connected to the database server. The server maintains a global lock table to ensure that clients cannot obtain conflicting locks on database objects simultaneously. When a client transaction requests database objects, the client's cache manager checks if these objects/locks are available locally. For data not available in the local cache, the client contacts the server and asks that the requested locks be granted and the corresponding objects be sent over. Once the necessary data becomes available, the client's CPU and available buffer space are used to carry out the necessary processing. The set of downloaded objects constitutes a local database that is stored in the client's memory as well as disk caches in an inter-transaction caching fashion. We support the client framework used in [43, 57, 59] where clients cache the locks for objects as well. Therefore, all future requests for the cached data (with the cached locks) can now be satisfied by the client locally. A client releases a lock on an object (and returns the object to the server, if necessary) if the server recalls it or if the client needs to create space in its local cache. Lock tables at the client and the server are updated accordingly. In addition, an invalidation mechanism is in place in order to avoid using cached but obsolete data objects [57]. We assume that clients can launch multiple concurrent transactions. Conflicts on objects requested by such local transactions are managed with the help of the client's own lock and object managers. Obviously, the lock that can be granted by a client's lock manager to a local transaction depends on the lock that the client has itself obtained from the server.

If a client's lock request on an object conflicts with the lock presently granted to other client(s) then the server sends callback messages to all such client(s) requesting that they release their locks as soon as possible [57]. If the object has been updated at a client then it is also required that the client returns the latest version of the object to the server. Once the object has been returned, the server grants the lock to the requesting client and sends the object over. In figure 1, consider a request by $Client_1$ for a SL on $Object_A$. $Object_A$



	A	B	C
1	NL	NL	NL
2	EL	SL	NL
3	NL	SL	SL
4	NL	SL	NL
5	NL	NL	SL

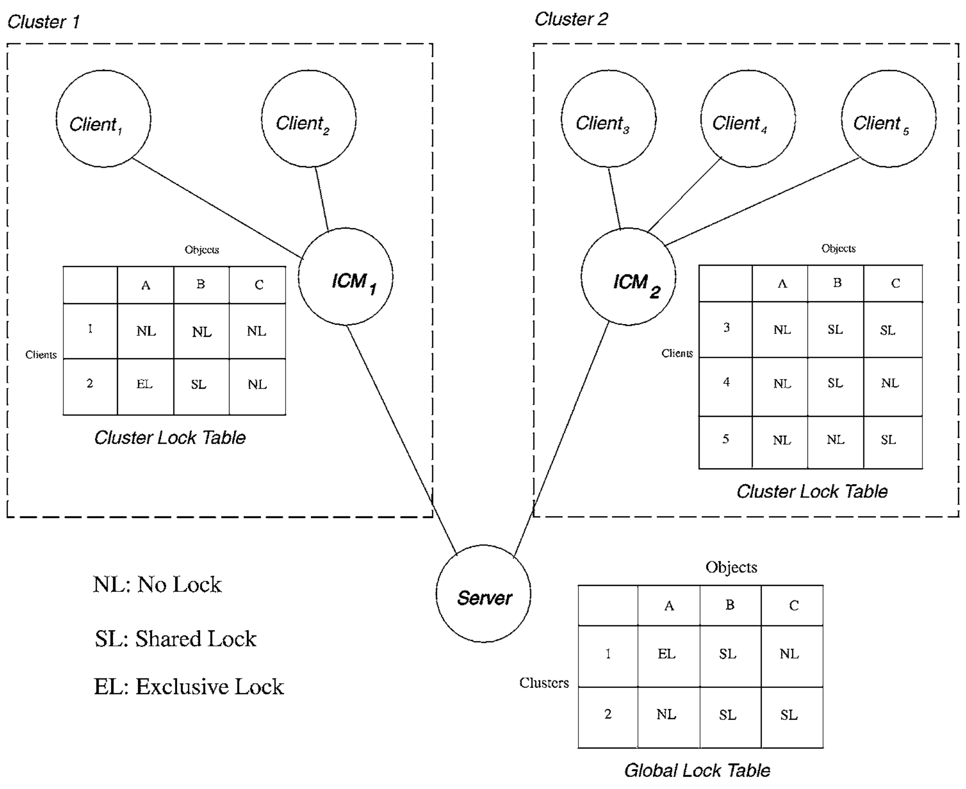
Figure 1. An example of object locking in FCS.

is exclusively locked by $Client_2$. Therefore, the server issues a callback request to $Client_2$ requesting it to return the object as soon as possible. Once $Client_2$ has finished its processing and returned the updated copy of the object releasing the EL, a SL is granted to $Client_1$ and $Object_A$ is shipped to it. Thus, FCS clients depend on the server as the only source of database objects that are not available in their local caches.

2.2. Logically clustered client-server architecture (LC-CS)

In LC-CS, clients that demonstrate similar object access patterns are logically clustered into groups. This contrasts with previous proposals where clustering has been performed on the basis of the clients' physical locations and existing network topology [13, 48]. Each group of clients in the LC-CS is managed by an Intermediate Cluster Manager (ICM). ICMs cooperate with the main database server and therefore, unlike FCS, clients are not directly connected to the server. The resulting LC-CS is a three-tier architecture with the ICMs serving as mediators (see figure 2).

Similar to the FCS, database processing in LC-CS is performed at the clients only. However in LC-CS, the server does not maintain a lock table for all clients in the system. Instead, it only keeps a lock table that stores locks granted at the cluster level. ICMs keep track of the object and lock status for individual clients in their clusters. This two-level locking enables the system to minimize the server's concurrency processing as object



Clients	A	B	C
1	NL	NL	NL
2	EL	SL	NL

Cluster Lock Table

Clients	A	B	C
3	NL	SL	SL
4	NL	SL	NL
5	NL	NL	SL

Cluster Lock Table

Clusters	A	B	C
1	EL	SL	NL
2	NL	SL	SL

Global Lock Table

Figure 2. Hierarchical locking in LC-CS.

requests satisfied within clusters do not directly interact with the server's lock manager. The latter updates its lock table only when locks are granted to clusters. An example of this hierarchical locking schema is shown in figure 2. The server has granted an EL on $Object_A$ to $Cluster_1$. ICM_1 now has the authority to grant SL or EL requests on $Object_A$ to any client within the cluster. Therefore, if $Client_2$ requests a lock on $Object_A$ then ICM_1 can grant this request without having to contact the server.

When a client transaction requests data objects/locks, the client checks whether the requests can be satisfied from its local disk and memory caches. If an object is not available locally then the client dispatches a request for that object (and the appropriate lock) to its ICM. The method in which the ICM satisfies requests for data objects depends on the type of locks that have been requested (shared or exclusive). The steps taken by the ICM for a SL request are shown below:

(i) When an ICM receives an object request from a client, it looks up its cluster directory to see if another client within the same cluster has the object cached.
(ii) If the object is present at a client within the cluster, then the ICM requests that client to forward the object to the requesting client as soon as possible. In figure 2, we consider

a request by $Client_1$ for a SL on $Object_A$. This request is forwarded by the ICM_1 to $Client_2$. $Client_2$ will ship a copy of $Object_A$ to $Client_1$. Since $Client_2$ has an EL on $Object_A$, the object is forwarded to $Client_1$ once the transaction that is using it has committed; $Client_2$ downgrades its own lock on $Object_A$ to a SL. ICM_1 is informed of the successful forwarding operation so that it can update the cluster lock table/directory accordingly. At the same time, the server's lock table entry containing EL for ICM_1 needs to be downgraded to SL. Before this downgrading take place, the server receives an updated object from ICM_1.

(iii) If the object is not cached at any client in the cluster then the ICM contacts the server and requests to ship a copy of it. The server can grant this request immediately if no other cluster has locked the requested object in exclusive mode (in figure 2, consider a request by $Client_1$ for a SL on $Object_C$). Otherwise, the server issues a callback request to the ICM that has an EL on the object; that ICM issues a callback to the client that holds the EL. As soon as the client releases its EL and returns the object to its ICM, the ICM ships it to the server, the server ships it to requesting ICM, and that ICM sends it to the client. An example of this case would be a request by $Client_3$ for a SL on $Object_A$ (figure 2). In this situation, ICM_2 requests the server for the object. The server calls back the object from ICM_1. ICM_1 issues a callback to $Client_2$. Once $Client_2$ returns the object to ICM_1, ICM_1 forwards it to the server. At that point, the object can be finally shipped to $Client_3$.

The steps that are taken when a client requests an EL are analogous to the ones described above. The basic requirement is that before a lock can be granted, it is necessary for all conflicting locks to have been released.

In general, a client returns an object (and releases the lock on it) only when it receives a callback request for that object or when it needs to create free space in its cache. In either case, the client and the ICM have to ensure that the lock tables at the ICM and the server are updated correctly. The manner in which this is done depends on the type of lock that the client is about to release:

(i) If the object has been updated at the client (EL) then the latest version of the object is shipped to the ICM before it is purged from the client's cache. Once the ICM has received the updated page, the client deletes the object. The ICM passes on the updated object to the server and updates its lock table entry to indicate that no copies of the object are present in the corresponding cluster.

(ii) If the client has a SL on the object then there is no need to ship the object back to the server. The client only needs to inform the ICM that it is about to purge the object from its cache. If no other client in the cluster has cached the object then the ICM informs the server that the object is no longer present in its cluster.

From this discussion, we can see that the use of ICMs can offer several advantages over the two-tier FCS architecture:

(i) when the data are cached at some client(s) in the cluster, the object location directories maintained by the ICMs allow requests for data to be satisfied without the intervention of the server,

(ii) the ICM's cluster-wide lock table allows sharing of data in its cluster and also guarantees that data accesses are serialized, and

(iii) the detection of deadlocks is carried out in a distributed manner [10]. ICMs are responsible for serializable transaction processing and deadlock detection within their own clusters whereas the server is responsible for enforcing system-wide serializability. Wait-for graphs at both ICMs and the server are used to detect and resolve deadlocks at the cluster and server levels.

Since interaction with the server may not be necessary, the above benefits can contribute significantly towards reducing the load on the database server when the clients in each cluster have a considerable overlap in their data requirements. In such situations, the scalability of the server can be vastly increased compared to that of the FCS server. The resource requirements of the ICMs are not very demanding either. Relatively small and inexpensive machines can be utilized to provide the object directory services and concurrency control.

2.3. Extended logical clustered client-server architecture (Extended-LC-CS)

The Extended-LC-CS architecture is designed to improve upon the benefits provided by the LC-CS. In LC-CS, every request that is satisfied within the cluster requires three messages: object request from the client to the ICM, forwarding request from the ICM to an appropriate client, and forwarded object from that client to the requesting client. This is due to the absence of data/lock caching abilities at the ICM. If ICMs are given the ability to cache database objects as well, an important enhancement becomes possible. When a client requests a SL then after the object is fetched from the server, the ICM can store it in its own cache too. This allows the ICM to satisfy future SL requests by itself. Hence, it is possible for non-conflicting (SL-SL) object requests to be satisfied with only two messages, without the need for request/object forwarding.

This saving in the number of messages (and, consequently, blocking time) can become very significant if the sites in each cluster demonstrate a degree of locality in their accesses to the data. The ICM can now function as a downsized-server and a subset of the functionality of the database server can be replicated at the ICM. This allows clusters to operate independently of each other as long as their data requirements do not intersect. Therefore, in contrast to the LC-CS, ICMs in the Extended-LC-CS first examine their own cache to see whether the requested objects and locks are available. If so, the request can be satisfied immediately. Otherwise, the request is processed in a manner similar to the LC-CS. In figure 3, consider a request by $Client_2$ for a SL on $Object_C$. Since, $Object_C$ has not been exclusively locked by any other client, the server grants this request immediately and $Object_C$ is shipped to ICM_1. ICM_1 forwards the object to $Client_2$ and also stores it in its own cache. Now, a request by $Client_1$ for a SL on $Object_C$ can be granted by the ICM_1 on its own authority.

In a LC-CS cluster, when an object is dropped by all clients, the object has to be returned to the server if necessary, and the ICM is required to release the corresponding lock. This

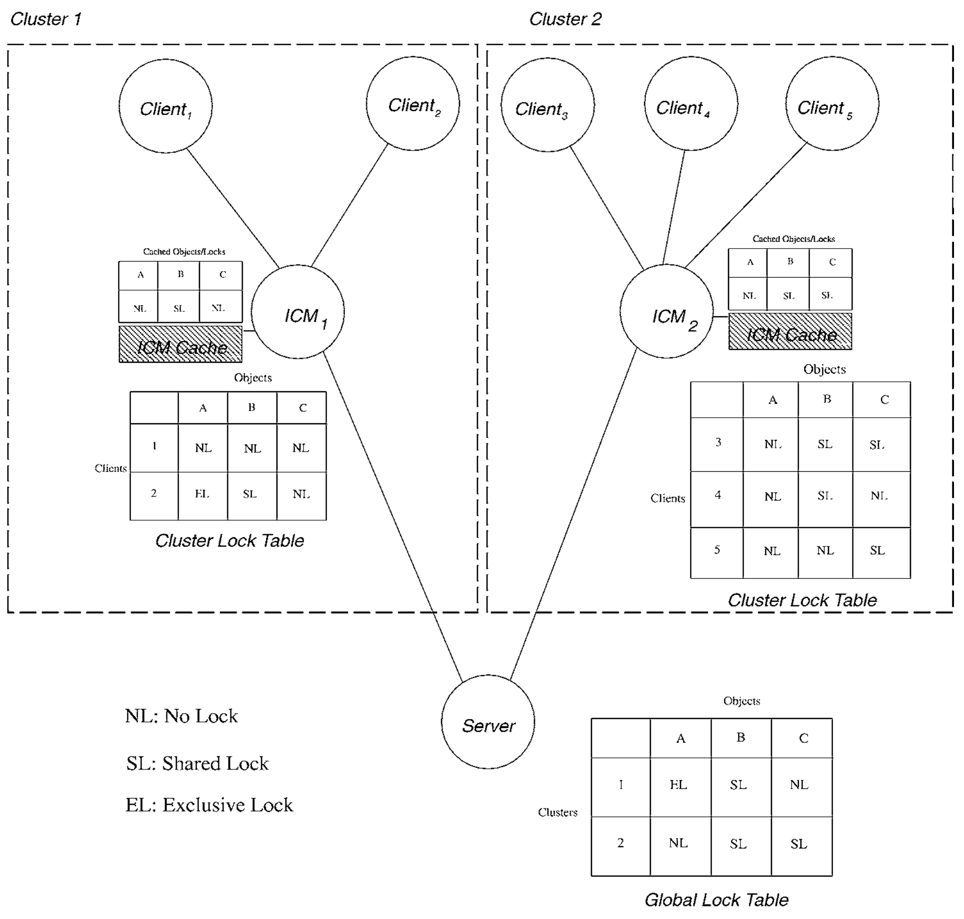


Figure 3. ICM caching and hierarchical locking in extended-LC-CS.

can create two adverse effects: (i) if an object, that is frequently accessed by the clients in a cluster, is dropped by that cluster then it may have to be re-fetched from the server in the near future. This causes increased transaction blocking times and also incurs additional messaging overhead, and (ii) the server is required to process each such request, therefore the scalability of the system (in terms of the total number of clients supported) is reduced considerably.

The Extended-LC-CS solves the above two problems effectively by incorporating caching abilities into the ICMs. The cache buffers available at an ICM are used to store objects that have been dropped from the caches of all the clients in the cluster. Now, the interaction with the server, that was necessary in LC-CS in the above cases, is no longer required. Future requests on data objects cached at the ICM level can be satisfied without server's assistance. The enhancements proposed in the Extended-LC-CS promise vastly improved response times for object requests, lower network utilization, and reduced server load. However, it is evident that these gains can only be substantial when the clients are clustered in a way that the sharing of data within the clusters is maximized. In fact, when client clusters are not defined satisfactorily, Extended-LC-CS can demonstrate a lower efficiency than even the basic FCS architecture. This is because the access serialization and messaging overheads are significantly higher for inter-cluster data requests than for intra-cluster ones.

2.4. *An analytical comparison between the two types of configurations*

In this subsection we perform an analysis of the two-tier (FCS) and three-tier (LC-CS and Extended-LC-CS) architectures based on the probability of object request satisfaction at the clients, and within the clusters. Table 1 lists the used variables and their descriptions.

First, we formulate an expression for the average time taken to satisfy an object request in the FCS architecture. This is derived from figure 4 which shows the flow of control for an object request. The label of each solid directed edge represents the cost (time delay) caused by the originating node of that edge.

Table 1. Variable definitions.

Variables	Descriptions
Probabilities	
P_{chit}	Prob (hit in a client's memory space)
P_{hit}	Prob (hit in a server's memory space)
Time Delays	
$CComp_{s-msg}$	Time for handling a message for a request in a client
$CRead_{obj}$	Time for reading from client's disk to memory
$CWait_{disk}$	Delay in a client's disk queue
CID_{search}	Time for searching a directory in a ICM
$IComp_{s-msg}$	Time for handling a message for a request in a ICM
$IWait_{service}$	Delay in an ICM's queue waiting a service
$LCSWait_{service}$	Delay in a server's queue in LC-CS
$RBuf_{search}$	Time for searching a cache buffer in a remote client
$RComp_{o-msg}$	Time for handling a message with an object in a remote client
$RWait_{service}$	Delay in the remote client's queue waiting a service
$SBuf_{search}$	Time for searching a memory space in a server
$SComp_{o-msg}$	Time for handling a message with an object in a server
$SRead_{obj}$	Time for reading from server's disk(s) to memory
$SWait_{disk}$	Delay in a server's disk queue
$SWait_{service}$	Delay in a server's queue in FCS
T_{o-msg}	Network delay for an object message
T_{s-msg}	Network delay for a request message

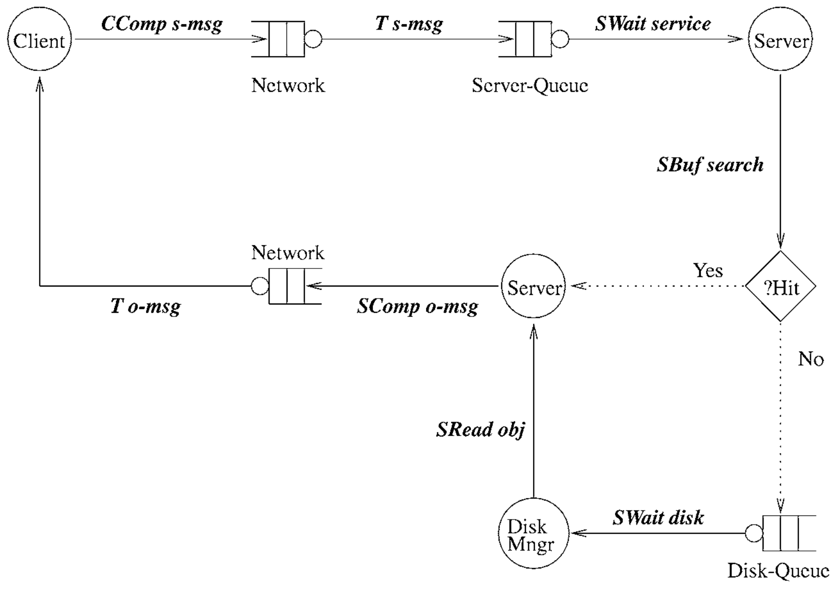


Figure 4. Control flow for an object service in the FCS.

Let t_{CC} be the average time for fetching an object from the client's local cache (disk and main memory), and let $t_{FCS-Server}$ be the average time taken by the server to satisfy an object request. If P_{C-FCS} is the probability with which a requested object is found in the client's cache, then the expected time taken for each object request to be satisfied is:

$$t_{FCS-Request} = P_{C-FCS} \times t_{CC} + (1 - P_{C-FCS}) \times t_{FCS-Server} \tag{1}$$

From figure 4, $t_{FCS-Server}$ can be expanded to:

$$t_{FCS-Server} = Computing_{FCS} + Network_{FCS} + Disk_{FCS} + Waiting_{FCS}, \tag{2}$$

where:

$$\begin{aligned} Computing_{FCS} &= CComp_{s-msg} + SBuf_{search} + SComp_{o-msg} \\ Network_{FCS} &= T_{s-msg} + T_{o-msg} \\ Disk_{FCS} &= (1 - P_{hit}) \times (SWait_{disk} + SRead_{obj}) \\ Waiting_{FCS} &= SWait_{service} \end{aligned}$$

In the three-tiered configurations, if an object is not available in the client's own cache then a request is dispatched to its corresponding ICM (see figure 5). Now, if the ICM cannot satisfy the request from within the cluster (either from it's own cache or from another client in the cluster), it sends a request for that object to the server. Let t_{ICM} be the average time

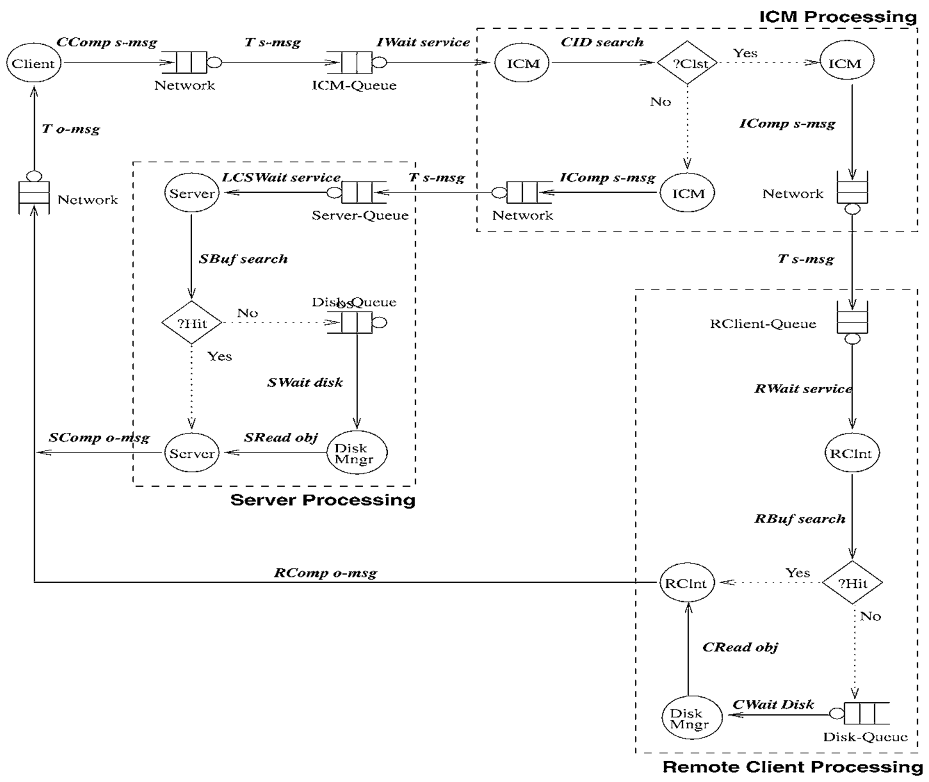


Figure 5. Process flow for an object reading with costs in the LC-CS.

for satisfying an object request within the cluster, and let $t_{LC-Server}$ be the average time for an object request serviced by the main database server. $t_{LC-Server}$ includes the time taken for request processing at the ICMs (to check whether the request can be satisfied within the cluster).

If $P_{C-LC-CS}$ is the probability that a client's request is satisfied locally and $P_{ICM-LC-CS}$ is the probability that the request sent to an ICM is satisfied within the cluster then the expected response time for a request is:

$$\begin{aligned} t_{LC-Request} = P_{C-LC-CS} \times t_{CC} + (1 - P_{C-LC-CS})(P_{ICM-LC-CS} \times t_{ICM} \\ +(1 - P_{ICM-LC-CS}) \times t_{LC-Server}) \end{aligned} \tag{3}$$

Here t_{ICM} can be expanded to:

$$t_{ICM} = Computing_{ICM} + Network_{ICM} + ClientDisk_{ICM} + Waiting_{ICM} \tag{4}$$

where:

$$\begin{aligned} Computing_{ICM} &= CComp_{s-msg} + CID_{search} + IComp_{s-msg} \\ &\quad + RBuf_{search} + RComp_{o-msg} \\ Network_{ICM} &= 2 \times T_{s-msg} + T_{o-msg} \\ ClientDisk_{ICM} &= (1 - P_{chit}) \times (CWait_{disk} + CRead_{obj}) \\ Waiting_{ICM} &= IWait_{service} + RWait_{service} \end{aligned}$$

$t_{LC-Server}$ can be also expanded such that:

$$t_{LC-Server} = Computing_{LCS} + Network_{LCS} + Disk_{LCS} + Waiting_{LCS} \tag{5}$$

where:

$$\begin{aligned} Computing_{LCS} &= CComp_{s-msg} + CID_{search} + IComp_{s-msg} + SBuf_{search} \\ &\quad + SComp_{o-msg} \\ Network_{LCS} &= 2 \times T_{s-msg} + T_{o-msg} \\ Disk_{LCS} &= (1 - P_{hit}) \times (SWait_{disk} + SRead_{obj}) \\ Waiting_{LCS} &= IWait_{service} + LCSWait_{service} \end{aligned}$$

In order to make a comparison between the expected response times at a client in the two architectures, we make the simplifying assumption that the cache hit ratios at clients are the same, i.e., $P_{C-LC-CS} = P_{C-FCS}$. This is not an unreasonable assumption as the cache-hit ratios at clients (in the two-tier or three-tier systems) that receive identical transaction streams (and, therefore, make the same object requests) will be very similar. Now, we use Eqs. 1 and 3 to derive an estimate for $P_{ICM-LC-CS}$, the probability that an object request is satisfied within the cluster, such that the overall average object response times in both architectures are the same. Equating response time equations 1 and 3, we get:

$$t_{FCS-Server} - t_{LC-Server} = P_{ICM-LC-CS} \times (t_{ICM} - t_{LC-Server}) \tag{6}$$

From this equation, $P_{ICM-LC-CS}$ can be formulated as:

$$P_{ICM-LC-CS} = \frac{(t_{FCS-Server} - t_{LC-Server})}{(t_{ICM} - t_{LC-Server})} \tag{7}$$

$P_{ICM-LC-CS}$ in Eq. 7 is the percentage of requests that need to be satisfied within the clusters in a three-tiered architecture to make the overall average object response times equal to that in the flat architecture. Using the detailed cost equations that are described above $P_{ICM-LC-CS}$ is calculated as:

$$P_{ICM-LC-CS} = \frac{E1}{E2} \tag{8}$$

where:

$$\begin{aligned}
E1 &= SWait_{service} - CID_{search} - IComp_{s-msg} - T_{s-msg} - IWait_{service} \\
&\quad -LCSWait_{service} \\
E2 &= RComp_{o-msg} + RBuf_{search} + RWait_{service} + (1 - P_{chit})(CWait_{disk} \\
&\quad +CRead_{obj}) - SCom_{o_m sg} - SBuf_{search} - LCSWait_{service} \\
&\quad +(1 - P_{hit})(SWait_{disk} + SRead_{obj})
\end{aligned}$$

As stated earlier, $P_{ICM-LC-CS}$ is the percentage of clients' object requests that are satisfied within their cluster. Equation 8 provides the lower bound value for $P_{ICM-LC-CS}$ that will make the object response times in the two architectures equal. A higher value of $P_{ICM-LC-CS}$ would imply lower response times in the three-tier configurations as compared to the FCS. From the equation for $E1$ and $E2$ we can see that the value of $P_{ICM-LC-CS}$ depends directly on the average queuing delays encountered in the main database servers in the two models, and on the queuing delay at the ICMs. Later in this paper, we show the average queuing delays measured during our experiments.

Logical client clustering, proposed in this paper, seeks to identify groups of clients that demonstrate similar database access behavior and place them together into clusters. If such logical clusters are well-formed then the sharing of data within clusters is increased. An increase in such data sharing leads to a corresponding increase in the value of $P_{ICM-LC-CS}$. In the next section, we describe the algorithms used to define client clusters based on their data access patterns, and the evaluation functions used to determine the quality of the clustering solutions.

3. Algorithms for logical client clustering

We first describe the input data, i.e, the format of the clients' database access pattern. Then, we describe two off-line clustering algorithms that are used to group clients with similar database access patterns into the same cluster. It should be noted that the optimal solution to this problem, based on identifying clusters that demonstrate the maximum access overlap, is NP-Complete [21, 44]. The first algorithm is based on greedy approach, and the second one is a genetic algorithm (GA) with a new encoding scheme and corresponding operators.

3.1. Input data representation

Without loss of generality, we assume that the database is a collection of uniquely identifiable objects. The object access patterns for individual clients can be created by monitoring data requests from the clients over a period of time.

The access pattern is represented by a collection of 0-1 bit strings, one for each client in the system. For n clients, let C be the set of n binary bit-strings, where the bit-string C_i represents the data access pattern for client i. Hence, for client i, $C_i = \langle b_i^1, \ldots, b_i^z \rangle$, where z is the number of objects in the database(s). The bits in each C_i are set using the rule:

$$b_i^j = \begin{cases} 1, & \text{if client } i \text{ accesses the } j\text{th data object} \\ 0, & \text{otherwise} \end{cases}$$

Using this representation of access patterns, the problem of finding a subset of clients with maximal common accesses becomes that of finding the largest set of overlapping 1's in the two-dimensional array C.

Obviously, in an environment where a client may access any database object, it is possible that after a long period of monitoring a great majority of bits for each client will be set to 1. In order to resolve this problem, we create bitmapped access patterns for each client during discrete time windows. The most recent access patterns are used to perform the clustering. In the next subsection, we describe a greedy algorithm to solve this problem.

3.2. Greedy algorithm for client clustering

In our first effort, we developed a single-pass greedy algorithm to determine client clusters. In a generic greedy algorithm, the choice made at each iteration is the one that is the best among the available options. Using this as the guiding principle, the algorithm we developed consisted of the following steps:

(i) If there are clients still unassigned to clusters then pick one of these clients randomly.
(ii) Add this client to the cluster that has the greatest database object access overlap with. If the client does not have a significant overlap with any cluster or if all the existing clusters are full, then a new cluster is created and the client is added to it.
(iii) Apply steps (i) and (ii) until all clients have been assigned to clusters.

The second step identifies which cluster a client should join. A bit-string is maintained for each cluster that represents the union of the sets of objects accessed by the clients in the group. This is used to help determine the affiliation of a client that needs to be clustered.

This algorithm is very fast (in real time) and it is easy to generate many possible solutions rapidly—by examining candidate clients in different orders. However, it has several disadvantages that make the generated solutions unsatisfactory: first, it searches for a solution only in the local solution space, i.e., all future solutions are in the immediate neighborhood of the current point. This makes it susceptible to local minima. Second, it also has no backtracking ability. Once a client has been assigned to a cluster, it cannot be later moved to another cluster. In order to improve upon this greedy algorithm, we address the client-clustering problem by using a genetic algorithm-based approach (GA). GA algorithms are known to be among the most promising of the evolutionary algorithms [24, 25, 27]. Genetic algorithms have received much attention as robust stochastic searching algorithms for various optimization problems [24] and have been frequently used in clustering problems that feature n-dimensional space input data [32, 54]. In [41], various evolutionary algorithms are investigated for data allocation among distributed server nodes, and a genetic algorithm approach shows superior performance to the other algorithms. After an initial evaluation of a number of optimization algorithms, we opted for a GA-based algorithm since, unlike other search-based techniques, a genetic algorithm starts with a large population of feasible solutions and performs a parallel search of the solution space [24]. This multi-modal approach makes the algorithm less likely to get caught in local minima, and ensures a more comprehensive search of the solution space. Furthermore, since new solutions are generated using probabilistic transition rules instead of deterministic procedures, a much more varied set of feasible solutions can be generated without resorting to exhaustive techniques.

3.3. The used genetic algorithm

To find a best-possible solution, we apply a GA to the above problem using two separate evaluation functions to generate succeeding generations of solutions. Using an appropriate measure to judge the quality of a generated solution is very important if the GA is to converge upon the best achievable solutions. The key objective of our evaluation functions is the maximization of potential common access pattern among clients as well as the minimization of the number of clusters and the overlapping access pattern among clusters. Our proposed functions, termed *IntraC* and *InterC*, are described below.

– *IntraC*: Clients in a group should have a very high percentage of common data accesses so that most object requests can be satisfied within the cluster. *IntraC* is a measure of the common data accesses of the clients in each cluster, taken over all clusters. Hence, for a generated solution consisting of k clusters:

$$IntraC = \sum_{i=1}^{k} \left(\frac{O_{Cluster_i}}{DBSIZE} \right)^2 + \frac{1}{k^2} \tag{9}$$

where $O_{Cluster_i}$ is the number of objects accessed in common by clients in $Cluster_i$ and *DBSIZE* is the size of the database. The ratio of $O_{Cluster_i}/DBSIZE$ represents the measurement of the quality for a single cluster (i.e., cluster cohesiveness). The evaluation function also encapsulates the number of clusters generated as a measure of quality of the clustering solution. This is an important parameter as it restricts the number of clusters generated. Without this restriction, the GA can generate solutions with arbitrarily large number of clusters thereby increasing the cost of implementing the ICM layer.

– *InterC*: Inter-cluster data accesses need to be as few as possible. For every object request that necessitates lock callbacks and releases across clusters the LC-CS and Extended-LC-CS systems incur a very high overhead. *InterC* is the percentage of inter-cluster data accesses made by the clients in all generated clusters. Therefore, for a generated solution consisting of k clusters:

$$InterC = \frac{\sum_{i=1}^{k} \frac{M_i}{N_i} \times c_i}{\sum_{i=1}^{k} c_i} \tag{10}$$

where N_i is the total number of objects accessed by clients in $Cluster_i$, M_i is the number of objects accessed by clients in $Cluster_i$ that are also accessed by clients from other clusters, and c_i is the number of clients in $Cluster_i$. The ratio of M_i/N_i indicates the probability of inter-cluster operations. The c_is are used to weight the evaluations of individual clusters so that a cluster with a larger number of clients will contribute more to the overall average.

We combine these two evaluation functions into a single metric f that gives equal weightage to both of them

$$f = \frac{1}{2}(IntraC + (1 - InterC)) \tag{11}$$

The $(1 - InterC)$ is necessary to convert $InterC$ from a minimizing function to a maximizing function. The sum of two factors is averaged so that the final value ranges between zero and one. The f is used in the GA to judge the fitness of the chromosomes in each generation.

In a GA, characteristics of the solution are represented as the genes of a chromosome. In our implementation, each chromosome is stored as a linked list of genes. Each gene represents one cluster, and the complete chromosome identifies one clustering solution. As depicted in figure 6, an initial population of chromosomes is first generated. For this, we use the greedy algorithm described earlier. The size of the initial population is chosen to be large enough so that the formation of many different chromosomes is feasible (one hundred is considered to be a reasonable size for the initial population [25]. The fitness of each chromosome in the population is calculated according to the evaluation function f (Eq. 11). From this initial population, the next generation of chromosomes is created by applying the laws of natural selection, i.e., chromosomes with better fitness have a higher probability of being selected for the next generation.

Using the chromosomes selected for the new population, we select the candidates for recombination arbitrarily. From this pool of candidate "parents", we select pairs of chromosomes randomly and generate their "children" by recombining the genes of the two parents. This enables the GA to search the unexplored area in the solution space. At the moment of recombination, a number of genes in each parent are selected for crossover, and the chosen genes are exchanged between parents generating two children. This GA recombination process is similar to the reproduction process in nature. The two parents are then replaced by the child chromosome in the population.

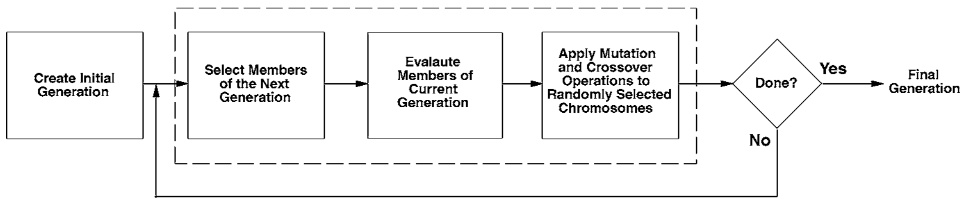

Create Initial Generation
Select Members of the Next Generation
Evalaute Members of Current Generation
Apply Mutation and Crossover Operations to Randomly Selected Chromosomes
Done?
Yes
Final Generation
No

Figure 6. Structure of the genetic algorithm.

Subsequently, genes from randomly picked chromosomes are mutated randomly by small amounts. In a similar fashion to the random selection and recombination operations discussed above, the GA's mutation operation makes the evolution among the candidate solutions possible. The frequency of mutation determines the degree of inheritance from the previous generation. If this frequency is too high, even chromosomes of very good quality cannot be inherited to the next generation and the GA becomes similar to one of the stochastic methods [24]. In contrast, if frequency of mutation is too low, the quality of chromosomes depends on the recombination process and GA appears like one of the deterministic approaches [24]. In summary, succeeding generations of chromosomes are generated either for a fixed number of generations or until a solution with the required fitness is found. In our experiments, we ran the GA until two hundred generations had been created and then picked the best available solution.

The GA has the disadvantage that it cannot prevent the creation of very large clusters if a great number of clients demonstrate very similar data access behavior. This can be a problem because now the ICM corresponding to that cluster may become a bottleneck. In these cases, an additional stage of processing is necessary. This extra processing involves running the GA again on the clients in the large cluster under consideration. However, now only the *InterC* evaluation function is used as we only want to minimize inter-cluster accesses among this subset of the original clients.

3.4. An example

The logical clustering solutions generated by 12 generations of the GA are shown in figure 7. The input to this example were the database access patterns of 10 clients. These

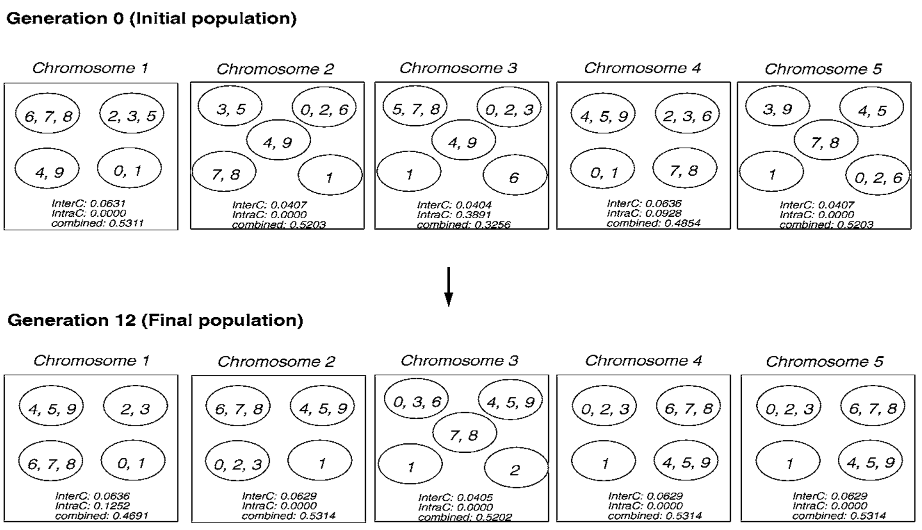


Figure 7. A sample run of the genetic algorithm with 10 clients.

access patterns were created randomly for a database containing 10,000 objects. The initial population of chromosomes was generated using the greedy maximal overlap algorithm and is shown as Generation 0 in figure 7. Here, we can see that Chromosome 1 is rated as the best solution and Chromosome 3 is the worst.

After 12 generations of recombinations and mutations, we can see that the chromosomes are converging towards a single optimal solution (according to the combined measure). Three of the five chromosomes are identical with a combined evaluation value of 0.5314. After a few more generations, all the chromosomes will suggest the same clustering solution. Since the GA is not an unimodal optimization algorithm, it can be seen that the final population also contains several unacceptable clustering solutions. This is, in fact, an advantage of a GA. The multi-modal search makes a GA less likely to get stuck in local optima.

4. Experimental evaluation

In this section, we describe the experimental evaluation of the three architectures and detailed prototypes that run in a network of workstations. By varying the number of clients and database access patterns, we have examined their performance indicators and scalability.

4.1. Methodology

Our test-bed consists of six Sun workstations running Solaris 7 and connected by a 10 Mbps LAN Ethernet. In our experiments, the database server ran by itself on one workstation, while the ICMs and the clients were equally distributed over the remaining machines. The values for key database system related parameters are shown in Table 2.

We have used the Paged File (PF) layer to manage object database systems at the clients, ICMs and the server [54]. The PF layer incorporates the functionality needed to maintain a page buffer in memory and also writes updated pages back to disk when necessary. The database consists of 10,000 objects with the size of each object as 256 bytes. We chose

Table 2. Key database-related parameters.

Parameter	Value
Database size	10,000 objects
Server main memory size	2,500 objects
ICM main memory size	500 objects
ICM disk capacity	500 objects
Client disk cache size	200 objects
Client memory cache size	100 objects
(Minimum, maximum) number of objects Accessed by a transaction	(1, 10)

this object size for experimental convenience only, and our sensitivity analysis shows that using objects of a larger size (up to 4 Kb) does not significantly affect the trends in our experimental results. Without loss of generality, we assume that exactly one object is stored in each page of the PF layer databases.

Communication between the clients and the server, and ICMs (in LC-CS and Extended-LC-CS), was done using TCP sockets. In our prototypes, the servers, clients and ICMs are designed to be connection-oriented, i.e., connections established at the beginning of the experiment are maintained for the duration of the experiment. Clients in FCS are connected directly to the database server. In LC-CS and Extended-LC-CS, clients are connected to the ICM corresponding to their cluster which, in turn, is connected to the server. In order to transfer data among clients in the most efficient manner, we have opted to use a specialized "directory server." The goal of the latter is to receive data objects from clients and forward them to their intended recipients without maintaining socket connections among all pairs of sites at all times. There are two benefits in using such a service: the significant time delay (in the order of 1 sec) that would incur in establishing and closing socket connections for each data transfer is avoided and the maximum number of open socket connections is linearly proportional to the number of clients. The logical network topologies for the FCS and the LC-CS/Extended-LC-CS implementations are shown in figure 8(a) and (b) respectively.

In FCS, the server processes all requests for data objects/locks from the clients. It maintains an up-to-date lock table and resolves all concurrency issues including deadlock detection (which is done using a graph-based deadlock detection algorithm [12]). In order to do this efficiently, we have designed the server, the clients and the ICMs as multi-threaded processes. The FCS server assigns one thread to each client in the system. This thread is responsible for handling all future interaction with that client. The multi-threaded implementation allows client data requests to be satisfied as efficiently as possible. In the two LC-CSs, the server interacts with a number of ICMs by designating a thread for each one of them. Similarly, each ICM executes one thread for every client in its cluster. We implemented these packages using the Solaris thread library. Synchronization between multiple

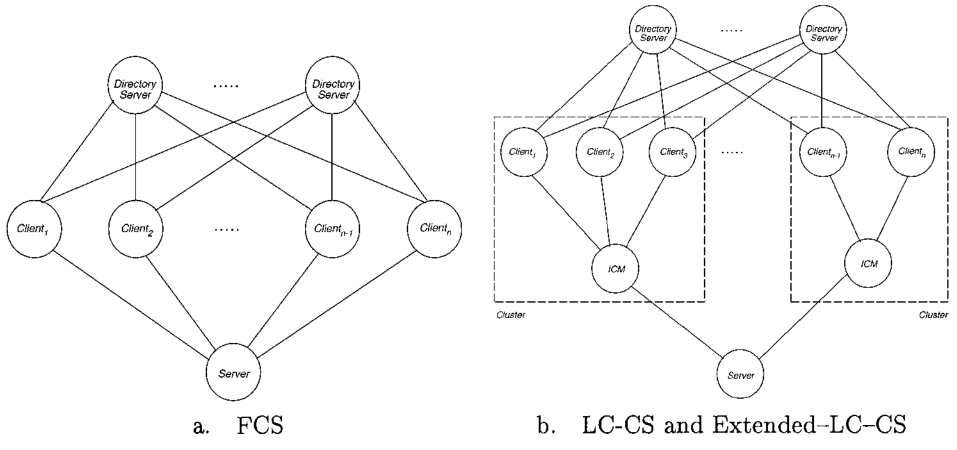


Figure 8. Logical network topologies for the implementations.

threads, when accessing global shared variables, was performed using the available mutual exclusion (mutex) primitives.

Clients may generate multiple concurrent transactions. Transaction arrivals at each client are generated as a Poisson process with a fixed inter-arrival mean of 1.5 seconds. The CPU processing time for each transaction is generated using an Exponential distribution with an average of 0.5 seconds. The transaction processing load on the server is varied by increasing or decreasing the number of clients. Each transaction requests multiple database objects following a uniform distribution between one and ten objects. It is assumed that a fraction of the objects accessed within a transaction will be modified. Modified objects ultimately create I/O write operations. Once objects become available, along with their appropriate locks, transactions are being executed as lightweight processes. If the required data is locally available, then it is locked by the transaction and is brought into the client's buffer memory. If the object is updated, then it is marked as dirty so that it is written back to the client buffers when the transaction commits.

The percentage of transaction objects that are modified is varied according to the workload in use. These workloads have been designed to compare the efficiency of the logically clustered schemes with that of the flat client-server architecture. We have used two types of workloads based on the HOTCOLD scenario [8] and a variation of the producer/consumer (also known as FEED) scenario [8, 14].

In the general HOTCOLD scenario, every client accesses mostly a specific range of the database (i.e., a "hot-spot" region) while rarely works with the remaining objects (i.e., the "cold" dataspace). In our case, we designate fifty disjoint hot-spots whose individual size is 1% of the database size. Clients access hot-spots 90% of the time and cold space only 10% of the time. We adopt two workloads that differ in the way client hot-spots are selected.

In the first HOTCOLD-based workload, each client randomly selects up to five hot-spots. We call this database access pattern *Hotspot-Scattered*. In the second HOTCOLD-based pattern, the set of hot-spots is divided into ten distinct subsets. Each client selects one of the ten such subsets for 90% of its I/Os and the remaining requests (10%) go anywhere in the cold section of the dataspace. In particular, the first 10% of the clients select their hot-spots between the range zero to four, the next 10% clients choose theirs between the range five to nine, and so on. We term this access workload *Hotspot-Concentrated*.

The goal of the above two workloads is to investigate the scalability of the three architectures in light of possibly favorable clustering. Since client object accesses are directed to specific ranges, clustering algorithms are able to group sites that work with similar hot-spots together. In these two workloads, the fraction of overall modified objects remains at a moderate level (i.e., 5% of accessed objects are modified) which is representative of everyday database processing [61].

Next, we examined the behavior of the three architectures in a producer/consumer (FEED-based) setting. A producer is a client that modifies half (50%) of its accessed objects. A consumer site simply reads (consumes) objects. In this type of workload, we assume that 10% of clients involved are producers.

The interaction between consumers and producers gives rise to two possible workloads. In the first, a subset of consumer clients and a producer client access a common subset of hot-spots. We call this database access pattern *Optimistic-FEED* as we conjecture that a

Table 3. Experiments.

	Experiment 1	Experiment 2	Experiment 3	Experiment 4
Database access pattern	Hotspot-Scattered	Hotspot-Concentrated	Optimistic-FEED	Pessimistic-FEED
Update selectivity	5%	5%	50%	50%

clustering algorithm will be able to group each producer and its corresponding consumers together. In the second producer/consumer workload, a producer accesses a set of randomly selected hot-spots (up to five). Similar to the first FEED-based workload, groups of consumer clients are set to access a common subset of hot-spots. We term this workload *Pessimistic-FEED*. Even a very effective clustering algorithm will be unable to create groups of producers and consumers whose inter-cluster data accesses are minimized. In both FEED-based workloads, 90% of the object accesses made by each client are directed to its designated hot-spot(s). Table 3 summarizes the performed experiments.

4.2. Experimental results

For all sets of experiments, we used the greedy and genetic algorithms to create client clusters. Both algorithms try to colocate clients that demonstrate common data access behavior. In addition, the optimization function used in the genetic algorithm also tries to reduce the occurrences of inter-cluster data accesses. We observed that the evaluation function for the solution generated by the genetic algorithm is significantly better than that of the greedy method. More specifically, in clustering with one hundred clients using a *Hotspot-Scattered* workload, the combined evaluation function for the GA was 0.563 whereas for the greedy algorithm it was 0.462. The percentage of accesses that were inter-cluster were 2.2% and 6.9% for the GA and greedy algorithm respectively.

In Table 4 we show the hot-spots accessed by 20 clients (due to space limitation) in a randomly generated *Hotspot-Scattered* workload. For this access pattern, the clustering solutions generated by the greedy algorithm and the genetic algorithm are shown in Table 5. From this example, it can be easily confirmed that the GA does better as it successfully clusters clients that access common database hot-spots. The percentage of inter-cluster accesses (of all data accesses) is 1.7% in the clustering generated by the GA while it is 4.1% in the greedy algorithm's solution.

Below, we present our results for the four experimental sets. We consider three key parameters that serve as indicators of the transaction processing performance of the three architectures: average transaction turnaround time, average object response time, and average queuing delay at the server for client object requests. In each graph, we present the results for the FCS and the two LC-CS architectures. We show the results for the clustering solutions generated by the genetic algorithm as well as the greedy one. The execution time of each experimental set is approximately 10 hours.

Table 4. Clients' hotspot accesses (20 clients, Hotspot-Scattered).

Client	Hot spots accessed	Client	Hot spots accessed
0	49	10	11, 13
1	14	11	1, 18, 35
2	46	12	4, 7, 11
3	15, 20, 34	13	8, 14, 39
4	21, 38	14	41, 47
5	45	15	29
6	39	16	6, 41
7	16, 20, 41	17	23, 26, 35
8	1	18	18, 33
9	5	19	11, 43

Table 5. Clients' clustering solutions (20 clients, Hotspot-Scattered).

	Greedy algorithm	Genetic algorithm
Clients in cluster 0	0, 1, 2, 3, 4, 5, 6, 7, 11, 14	0, 1, 5, 6 10, 11, 12, 15, 18, 19
Clients in cluster 1	8, 9, 10, 12, 13, 15, 16 17, 18, 19	2, 3, 4, 7, 8, 9, 13, 14, 16, 17
Evaluation function f	0.582	0.667

4.2.1. Experiment 1: Hotspot-scattered. Figure 9 shows the average transaction turnaround times in the three architectures. In FCS, the average turnaround time increases almost linearly as the number of clients increases. The increase in turnaround times in the two clustered architectures is much more gradual. Very similar trends can also be seen in the average object response times (figure 10) and the average request queuing delays at the server (figure 11). The rate at which the average object response time increases for FCS is much greater than in the LC-CS or Extended-LC-CS. The observed trends are a direct result of the *Hotspot-Scattered* workload.

The clustering generated by the genetic or the greedy algorithm have a very small percentage of inter-cluster accesses. This ensures that most object requests are satisfied within the cluster and the load on the server is significantly reduced. Hence, increasing the number of clients does not linearly increase the load on the LC-CS server. However, this is not the case in the FCS configuration. Since all requests have to go through the server, any increase in the number of clients implies a corresponding rise in the load over the server. The additional overheads of maintaining global data consistency and concurrent accesses also increase rapidly in FCS.

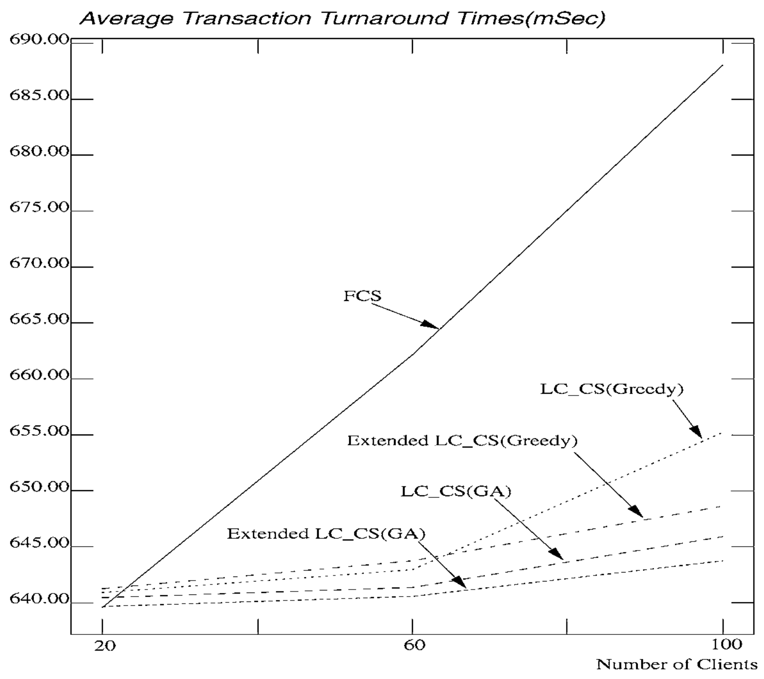


Figure 9. Experiment 1—Hotspot-Scattered: Average turnaround times for client transactions (milliseconds).

In figure 12, we can see that in the presence of 100 clients in LC-CS (Greedy) approximately 15% of all object requests are satisfied within the cluster using object forwarding. The Extended-LC-CS configuration has the added feature of the ICM cache space. Objects that have been purged by all clients from their own caches are stored at the ICMs. This makes it possible for the ICM to satisfy requests for such objects without referencing the server. For 100 clients in Extended-LC-CS, 3.9% of all object requests are satisfied from the ICMs' caches. This is in addition to the object requests satisfied from the caches of other clients within the same cluster.

4.2.2. Experiment 2: Hotspot-concentrated. The results for the second set of experiments are shown in figures 13–16. In the *Hotspot-Concentrated* workload each group of clients shares a disjoint subset of hot-spots. Therefore, clustering solutions are easier to identify and the difference in the quality of the clustering solutions generated by the genetic and greedy algorithms is reduced. Here, there is an even smaller percentage of inter-cluster accesses as compared to the clustering solutions for the *Hotspot-Scattered* workload.

The overall performance trends are similar to those derived in the previous experiment. In the clustered architectures, the overall average object response time is considerably shorter than that in FCS. For most requests, ICMs can locate an object copy within their boundaries. Thus, client requests can be satisfied without the intervention of the server. Only in the case of lock conflicts between clusters is the server required to perform object callbacks and

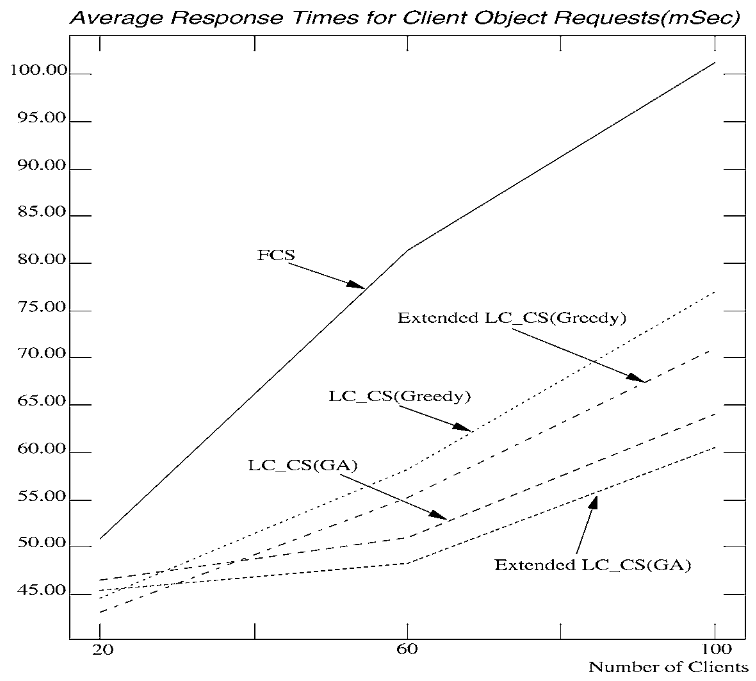


Figure 10. Experiment 1—Hotspot-Scattered: Average response times for client object requests (milliseconds).

update its lock tables. For instance, in the Extended-LC-CS (GA) architecture and for 100 clients, inter-cluster lock conflicts occurred in only 5.9% of all object requests.

Comparing the results of figures 9, 10 and figures 13, 14, the observed turnaround and object response times are longer in the second experiment. This is especially evident when 100 clients participate. Longer times are attributed to the much stronger contention generated by *Hotspot-Concentrated* as client access "contiguous" hot-spots. Serving such conflicting object accesses involves longer processing times.

4.2.3. Experiment 3: Optimistic-FEED. This set of experiments uses the *Optimistic-FEED* workload. Due to the distinct access behavior of each set of clients, the two clustering algorithms are able to easily identify good groups. In fact, both the genetic algorithm and the greedy algorithm give identical clustering results. For all numbers of clients, producers and their corresponding consumers are colocated in the same cluster.

The results are shown in figures 17–19. The graphs indicate that even with the extremely high update rates at the producers (50% of all object accesses are updates), LC-CS and Extended-LC-CS show gradual increases in both transaction turnaround time and object response time. Usually, when the update ratio is very high, it is expected that the transaction turnaround time and object response time will increase very rapidly as the number of clients increase. This is due to the resulting object recalls and blocking delays at server. However,

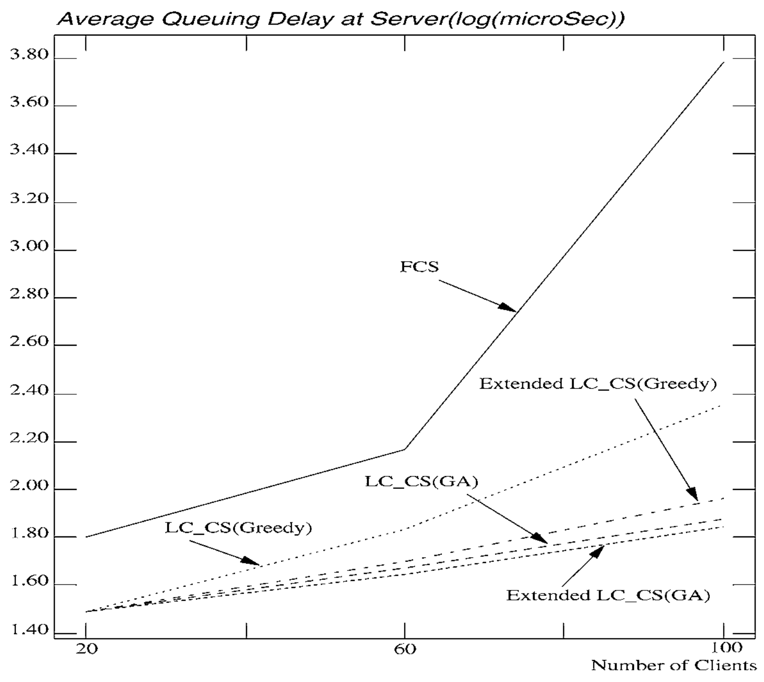


Figure 11. Experiment 1—Hotspot-Scattered: Average queuing delay at server (Microseconds—log scale).

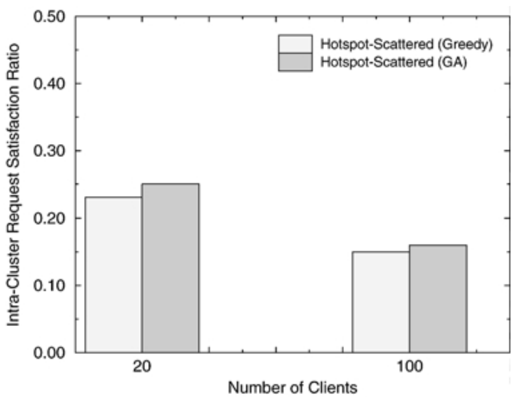


Figure 12. Experiment 1—Hotspot-Scattered: Average cluster-level object hit ratios for the LC-CS.

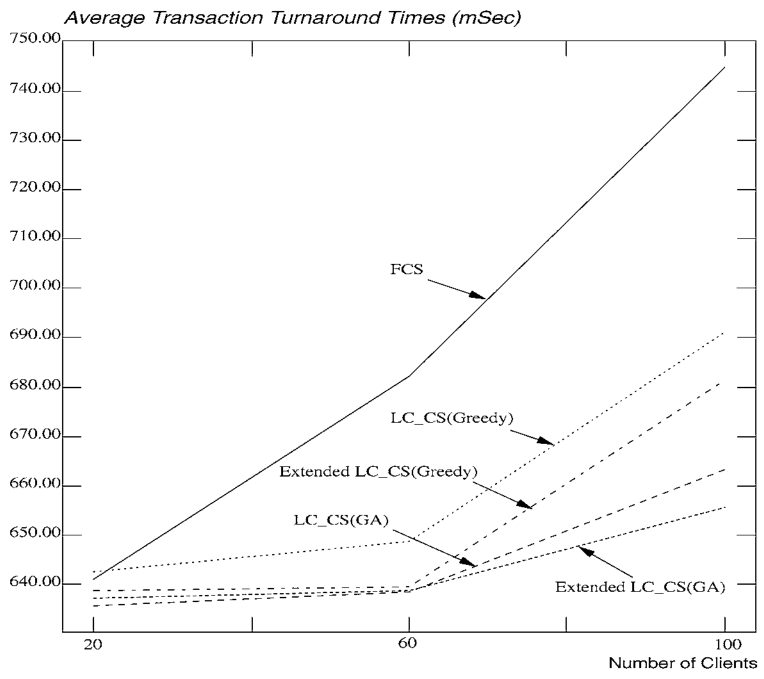


Figure 13. Experiment 2—Hotspot-Concentrated: Average turnaround times for client transactions (msecs).

when updated objects are accessed only by clients within the same cluster, the blocking factor at the server can be avoided. This is clearly seen in figure 19. Thus, the *Optimistic-FEED* workload scenario is very amenable to a logically clustered architecture.

4.2.4. Experiment 4: Pessimistic-FEED. For this set of experiments we use the *Pessimistic-FEED* workload. In contrast with the *Optimistic-FEED* workload, here the producers access randomly selected hot-spots. Therefore, the clustering algorithms are unable to group producers and the consumers corresponding to each of them in the same cluster. Again, producers update 50% of their accessed objects. The results for this set are shown in figures 20–22.

The performance of LC-CS and Extended-LC-CS are either similar to or worse than that of FCS in most cases. The average turnaround times for FCS are better than in the clustered architectures as consumers are not located in the same cluster with their producers. When there is an inter-cluster access, at least two ICMs should be involved in the processing. In turn, this creates delays at the ICM level which cumulatively present long turnaround times for the LC-CS and the Extended-LC-CS configurations. The above is true in spite of the fact that for 100 clients the average FCS object response time is worse than the other two (i.e., see figure 21).

From all four sets of experiments, we observed that clustering the LC-CS and Extended-LC-CS architectures using the GA consistently results in superior performance than when

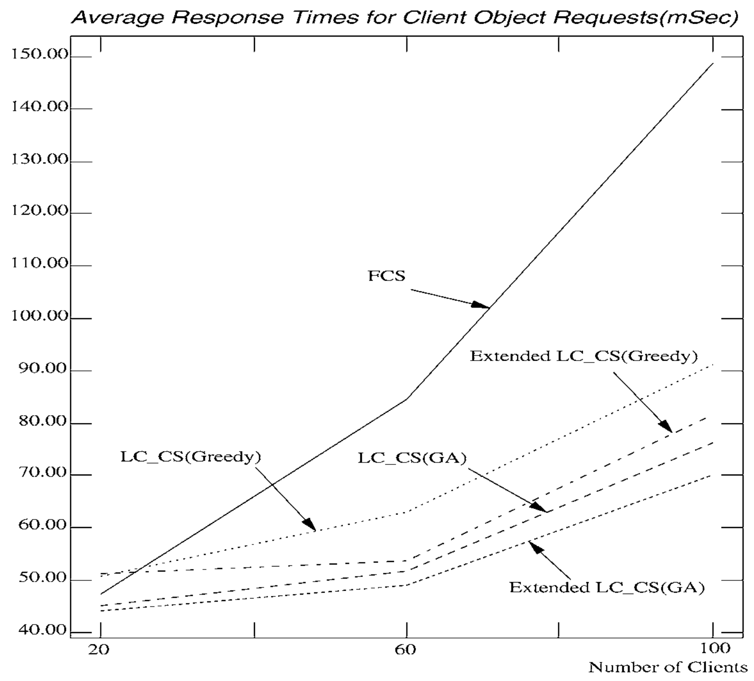


Figure 14. Experiment 2—Hotspot-Concentrated: Average response times for client object requests (msecs).

using the greedy technique. These results reflect the performance of the two algorithms in terms of their access similarity detection and the composition of well-formed clusters. However, it should be noted that there exists a tradeoff between the performance and the necessary CPU time and memory requirements of the two algorithms. In order to cluster 100 clients, the GA needs more than 7,000 seconds while the greedy requires only 80 seconds.

5. Related work

Most of the related work in the clustering of workstations/clients has been carried out in the area of distributed file systems (DFS). The Frolic DFS is an early representative of systems that allow replication of files over a number of sites [48]. Its main objective is to offer transparent file services over physically separate networks of workstations. Frolic targets environments with extensive file sharing but rare read/write conflicts across multiple clusters. Each cluster of workstations is coordinated by at least one server. A protocol for maintaining strong data consistency in a distributed file system was proposed in [57]. Here, although data files are replicated over multiple servers, no one server is held to be responsible for control over the entire file system. Instead, file-state information is stored at a majority of the servers that can create a consistent state of each file. The xFS system [13] advocates an environment for handling file requests. Here, files are moved away from the server to

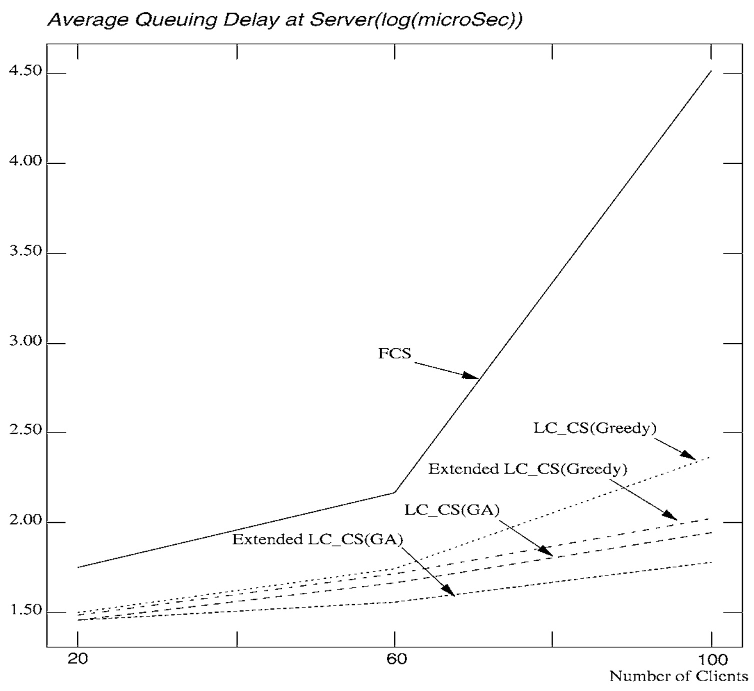


Figure 15. Experiment 2—Hotspot-Concentrated: Average queuing delay at server (Microseconds–Log scale).

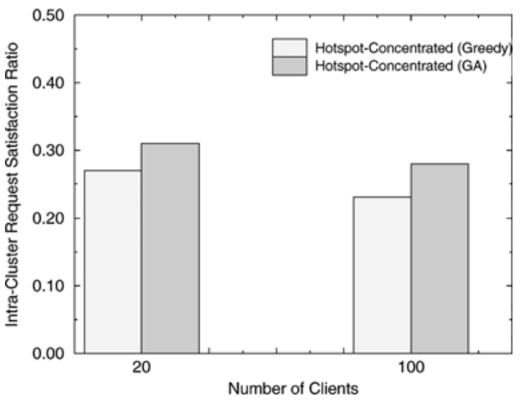


Figure 16. Experiments 2—Hotspot-Concentrated: Average cluster-level object hit ratios for the LC-CS.

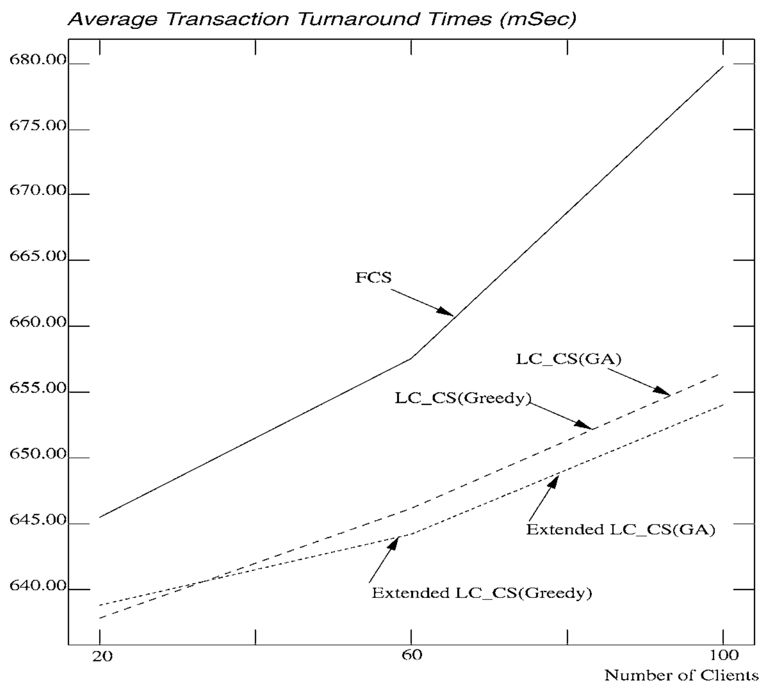


Figure 17. Experiment 3—Optimistic-FEED: Average turnaround times for client transactions (msecs).

the client sites. Therefore, the server is relieved from high loads usually observed is server-based file systems such as the AFS [49] and Sun NFS. The role of the server is to monitor the location of file copies within a group of workstations and forward file requests to appropriate sites. A hierarchical caching technique applicable to large-scale distributed file systems is discussed in [6]. The goal of this proposal is to ease central server contention by diverting file requests to clients that already maintain copies through a multi-level metadata structure. Our work differs from the approaches discussed above in that our approach dynamically creates client clusters, based on the clients' data access patterns, rather than on factors such as physical proximity.

In an early work on distributed databases, fragmentation techniques are used in order to split a database into partitions and place them at different sites with or without replication of data [42]. In [2], a model that optimally places relations in a network is presented. In the same work, it is shown that the problem of determining a non-redundant allocation is NP-hard. Vertical fragmentation for distributed database design is discussed in [47]. Clustering of database objects has been a popular area of research over the last few years. A number of techniques try to minimize the incurred I/O costs for mostly object-oriented databases [3, 58, 61]. The core idea is that objects frequently fetched together are grouped into a storage segment that caters for fast access. Efficient retrieval is feasible mostly sequential scans over data segments and avoidance of random movements of the disk head.

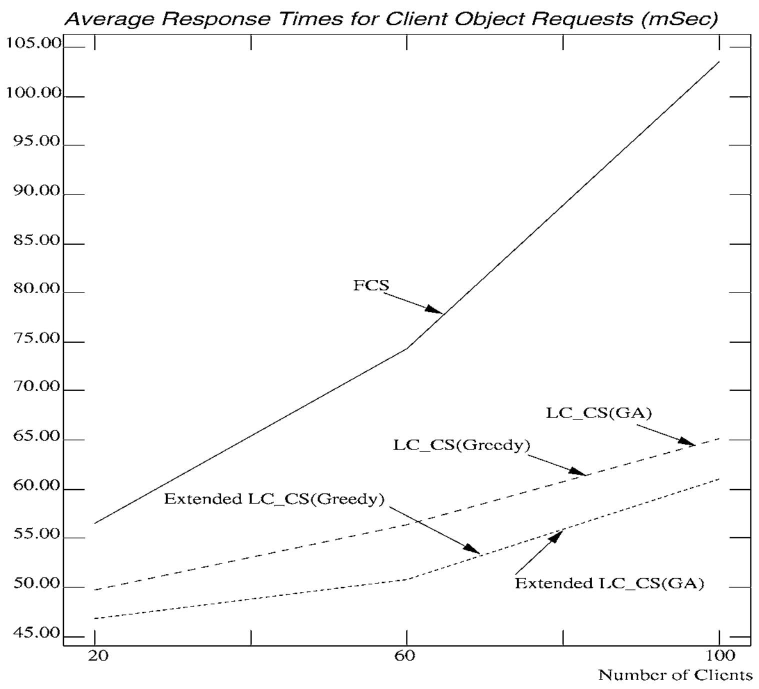


Figure 18. Experiment 3—Optimistic-FEED: Average response times for client object requests (msecs).

[22, 28, 38, 56]. In all above efforts, the fragmentation and object clustering techniques are performed at the storage level only. The same is the case with most commercial systems including ObjectStore [33], Versant [58], and the object-oriented application builder/suite Forté [7]. In this work, we cluster inter-networked systems at the level of sites as opposed to clustering data objects within storage devices used thus far.

Distributed caching techniques have also been proposed and implemented for the dissemination of information on the World-Wide Web (WWW). Here, the basic idea is to cache frequently requested web pages in locations, called proxy-servers, that are closer to interested clients/users [23, 36]. When a client requests a particular page, the request is first sent to the proxy server. If the page is available at the proxy server then the request is satisfied without having to contact the remote server at all. Using such caching, the load on web servers can be reduced, response times can be shortened, and the overall network bandwidth consumption can be optimized. An analysis of internet existing caching hierarchies and scalable caching techniques for reducing client response latency are discussed in [9, 19, 20, 37, 46, 53]. These techniques can be classified to those that are based on cooperating web-caching [9, 37] and those built around directory services of cached contents [19, 20, 46]. In contrast to the WWW, where the data is almost entirely read-only, the main focus of our work is on database systems where data updates and potential access conflicts are significant in terms of both size and frequency.

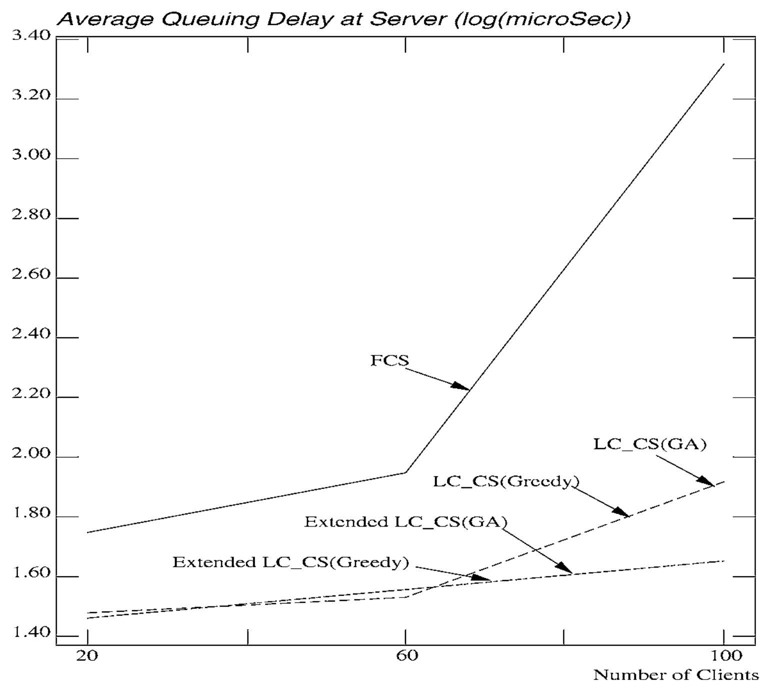


Figure 19. Experiment 3—Optimistic-FEED: Average queuing delay at server (Microseconds–log scale).

Most prior studies that deal with data access patterns aggregate multiple streams of requests from a population of users into a cumulative stream. In these efforts, the major concern is in understanding the group features of the workload generated by a client community so that hit-ratios for pages/objects can be increased in web-caches/proxies [18, 26, 31]. In contrast, our work treats every client access pattern individually in order to derive good client clusters.

6. Conclusions and future directions

In this paper, we have proposed an alternative to the conventional client-server database model that features client clustering and uses Intermediate Cluster Managers (ICM). Clients are logically colocated based on the similarity of their data accesses. Each such group is coordinated by an ICM that undertakes limited server tasks. Depending on whether there are caching capabilities at the ICM, two three-tier architectures are suggested: Logically-Clustered CS (LC-CS) and Extended-LC-CS. We have developed prototype packages that implement these two architectures as well as the conventional FCS configuration. To support three-tier architectures, we also developed two client clustering algorithms: one based on a genetic algorithm approach and the other based on a greedy heuristic technique. We have carried out a number of experiments under diverse workloads with all three packages and

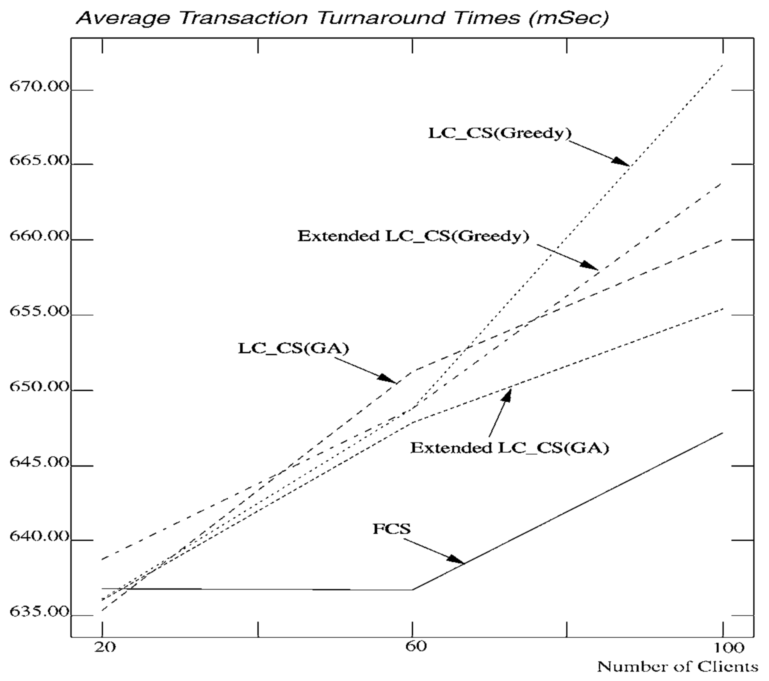


Figure 20. Experiment 4—Pessimistic-FEED: Average turnaround times for client transactions (msecs).

have experimented with two logical client clustering techniques. Our main results are:

– Given a set of database access patterns, it is possible to create clusters of clients that access common segments of the data space and reduce inter-cluster data accesses. This allows the introduction of an intermediate layer in the system hierarchy which satisfies the data requirements of clients within individual clusters.
– Well-formed clustering in LC-CS results in a substantial reduction in server load. This allows the system to scale-up to a much larger number of clients than that of the FCS architecture. ICMs can assist in increasing scalability without imposing significant penalties and overheads.
– System performance rates can be further improved by extending the capabilities of the ICMs to allow them to cache data/locks (Extended-LC-CS). The observed reduction in the message-passing and the corresponding transaction blocking is significant. However, this is not always the case. If the database access patterns do not permit a good client clustering formation, the FCS could demonstrate similar or even better performance levels. This proved to be the case for our *Pessimistic-FEED* workload.

In most database environments, it is to be expected that clients' data access requirements will change over time. Such behavior should be detected so that clients can be moved

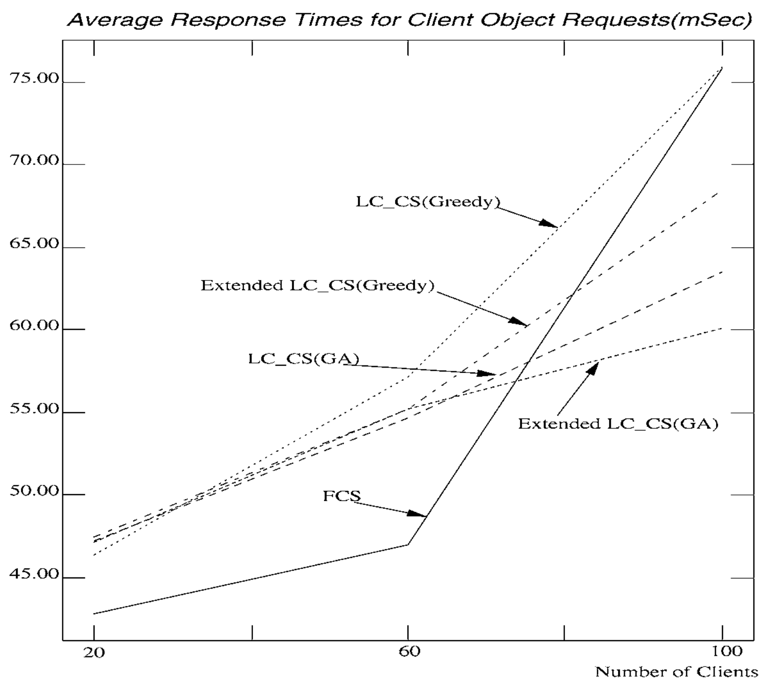


Figure 21. Experiment 4—Pessimistic-FEED: Average response times for client object requests (msecs).

to clusters that best match their new data requirements. We are currently exploring two approaches that can be used to re-assign clients to more appropriate clusters:

(i) Individual re-assignment: A client with deviating data access behavior is moved to a more appropriate cluster. To achieve this, clients' access patterns should be monitored, and occurring changes should be detected within acceptable deadlines. The change detection can take place at the ICM layer.

(ii) Global re-clustering: Current information regarding the data access patterns of all clients is used to re-generate the clustering formation. The data access patterns of each client from the most recent time windows are shipped from the ICMs to the server and the clustering algorithm is re-run to determine the new compositions of the clusters. The system can perform re-clustering periodically or within periods whose duration can be a function of important system parameters (including CPU server load, network traffic, length of ICM/server disk queues etc.).

In the future, we also plan to investigate issues related to database evolution and provide the features that allow our architecture to dynamically connect/disconnect and function with multiple database servers.

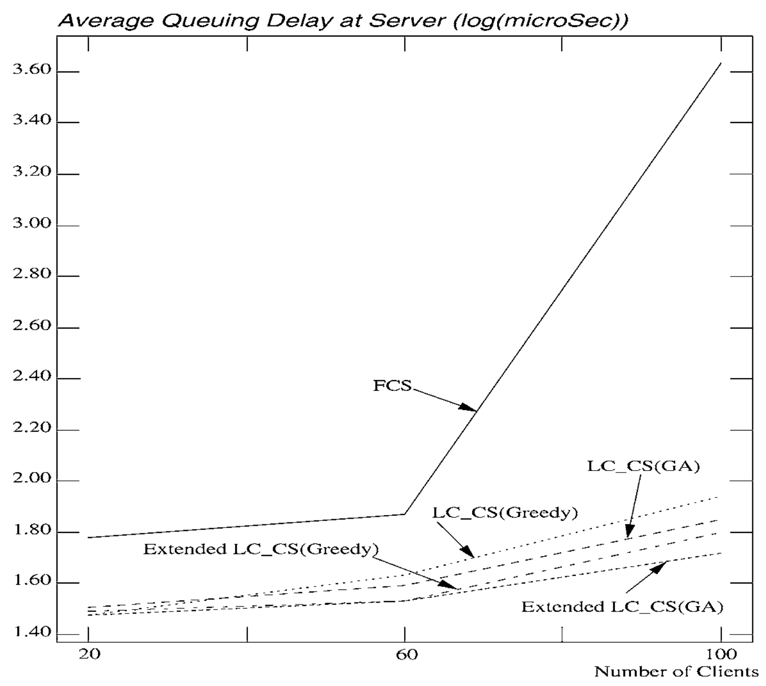


Figure 22. Experiment 4—Pessimistic-FEED: Average queuing delay at server (Microseconds–Log scale).

Acknowledgment

This work was supported in part by the National Science Foundation under Grant NSF IIS-9733642 and the Center for Advanced Technology in Telecommunications, Brooklyn, NY.

We would like to thank Maor Avidor and the reviewers for their careful and detailed comments that helped us significantly improve the presentation of this work.

References

1. J. Andrade, M. Carges, and M. MacBlane, "The TUXEDO System: An open on-line transaction processing environment," Data Engineering Bulletin, vol. 17, no. 1, 1994.
2. P. Apers, "Data allocation in distributed database systems," ACM-Transaction on Database Systems, vol. 13, no. 3, pp. 263–304, 1988.
3. J. Banerjee, W. Kim, S.-J. Kim, and J.F. Garza, "Clustering a DAG for CAD Databases," IEEE Transactions on Software Engineering, vol. 14, no. 11, 1988.
4. P. Bernstein, V. Hadzilakos, and N. Goodman, Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman, Reading, MA, 1987.
5. A. Biliris and J. Orenstein, "Object storage management architectures," in: Advances in Object-Oriented Database Systems, Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Kusadasi, Turkey, 1993.

6. M. Blaze and R. Alonso, "Dynamic hierarchical caching in large-scale distributed File Systems," in: Proc. 12th International Conference On Distributed Computing Systems, Yokohama, Japan, 1992.
7. P. Butterworth, "The resurgent mainframe and the future of distributed computing," Technical report, Forté Software Inc., Oakland, CA. White Paper on Forté Fusion Technologies available at *http://www.forte.com*, 1999.
8. M. Carey, M. Franklin, and M. Zaharioudakis, "Fine-grained sharing in a page server OODBMS," in: Proceedings of the ACM SIGMOD Conference, Minneapolis, MN, 1994.
9. A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell, "A hierarchical internet object cache". in: Proceedings of the USENIX 1996 Annual Technical Conference, San Diego, pp. 153–163, 1996.
10. R. Chow and T. Johnson, Distributed Operating Systems and Algorithms, Addison-Wesley Reading, MA 1997.
11. I. Chu and M. Winslett, "Choices in database workstation-server architecture," in: Proceedings of the 17th Annual International Computer Software and Applications Conference, Phoenix, AZ, 1993.
12. T. Cormen, C. Leiserson, and R. Rivest: 1990, *Introduction to Algorithms*. New York, NY: McGraw Hill .
13. M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson, "A quantitative analysis of cache policies for scalable network file systems," in Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems, 1994.
14. A. Delis and N. Roussopoulos, "Performance comparison of three modern DBMS architectures," IEEE–Transactions on Software Engineering, vol. 19, no. 2, pp. 120–138, 1993.
15. D. DeWitt, P. Futtersack, D. Maier, and F. Velez, "A study of three alternative workstation-server architectures for object oriented database systems," in Proceedings of the 16th International Conference on Very Large Data Bases, Brisbane, Queensland, Australia, pp. 107–121, 1990.
16. D. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A scalable and highly available web server," in Proceedings of COMPCON 1996, Forty-First IEEE Computer Society International Conference: Technologies for the Information Superhighway, Santa Clara, CA, 1996.
17. D. Dilts and W. Wu, "Using knowledge-based technology to integrate CIM databases," IEEE Transactions on Knowledge and Data Engineering, vol. 3, no. 2, pp. 237–245, 1991.
18. B. Duska, D. Marwood, and M. Freeley, "The measured access characteristics of World-Wide-Web client proxy caches," in Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS-97), Monterey, CA, 1997.
19. L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in: Proceedings of the ACM SIGCOMM'98 Conference, Vancouver, Canada, pp. 254–265, 1998.
20. S. Gadde, M. Rabinovich, and J. Chase, "Reduce, reuse, recycle: An approach to building large internet caches," in Proceedings of the 6th Workshop on Hot Topics in Operating Systems, Cape Cod, MA, 1997.
21. M. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness W.H. Freeman & Company, New York, NY, 1979.
22. G. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte, "Partition-based clustering in object bases: From theory to practice," in Proceedings of the International Conference on Foundations of Data Organization, Chicago, IL, 1993.
23. S. Glassman, "A caching relay for the World-Wide Web," in Proceedings of the First International World Wide Web Conference, Geneva, Switzerland, 1994.
24. D. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley Reading, MA, 1989.
25. J. Grefenstette, "Optimization of control parameters for genetic algorithms," IEEE Transactions on Systems, Man and Cybernetics, vol. 16, no. 1, pp. 122–128, 1986.
26. S. Gribble and E. Brewer, "System design issues for internet middleware services: Deductions from a large client trace," in Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS-97), Monterey, CA, 1997.
27. J. Holland, Adaptation in Natural and Artificial Systems, Ann Arbor, MI, University of Michigan Press, 1975.
28. S. Hudson and R. King, "Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system," ACM Transactions on Database Systems, vol. 14, no. 3, pp. 291–321, 1989.
29. A. Hurson, S. Pakzad, and J. Cheng, "Object-oriented database management systems: Evolution and performance issues," IEEE Computer, vol. 26, no. 2, 1993.

30. H. Ishikawa, Y. Yamane, Y. Izumida, and N. Kawato, “An object-oriented database system Jasmine: Implementation, application, and extension,” IEEE Transactions on Knowledge and Data Engineering, vol. 8, no. 2, 1996.
31. A. Iyengar, M. Squillante, and L. Zhang, “Analysis and characterization of large-scale web server access patterns and performance,” World Wide Web, vol. 2, nos. 1–2, 1999.
32. A. Jain, M. Murty, and P. Flynn, “Data clustering: A review,” ACM Computing Surveys, vol. 31, no. 3, pp. 264–323, 1999.
33. G. Jones, ObjectStore 6.0, Technical report, Object Design, Inc., 1999.
34. H. Kitagawa and N. Ohbo, “Design data modeling with versioned conceptual configuration,” in Proceedings of the 13th Annual International Computer Software and Applications Conference, Orlando, FL, September 1989.
35. A. Leff, P. Yu, and J. Wolf, “Policies for efficient memory utilization in a remote caching architecture,” Miami Beach, FL, December 1991.
36. A. Luotonen and K. Atlis, “World-Wide Web proxies,” in Proceedings of the First International World Wide Web Conference, Geneva, Switzerland, 1994.
37. R. Malpani, J. Lorch, and D. Berger, “Making World-Wide Web caching servers cooperate,” in Proceedings of the 4th International WWW Conference, Boston, MA, 1995.
38. J. McIver and R. King, “Self-adaptive, on-line reclustering of complex object data,” in Proceedings of the International Conference on Management of Data, ACM Press, Minneapolis, MI, 1994.
39. S. Milliner, A. Bouguettaya, and M. Papazoglou, “A scalable architecture for autonomous heterogeneous database interactions,” in Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, 1995.
40. C. Mohan and I. Narang, “ARIES/CSA: A method for database recovery in client-server architectures,” SIGMOD Record, vol. 23, no. 2, pp. 55–66, 1994.
41. M. Oates, D. Corne, and R. Loader, “Investigating evolutionary approaches for self-adaptation in the large distributed databases,” in Proceedings of the 1998 IEEE International Conference on Evolutionary Computation, Anchorage, AK, 1998.
42. M. Ozsu and P. Valduriez, Principles of Distributed Database Systems, Upper Saddle River, NJ, Second Edition, 1999.
43. E. Panagos, A. Biliris, H. Jagadish, and R. Rastogi, “Client-based logging for high performance distributed architectures,” in Proceedings of the 12th International Conference on Data Engineering, New Orleans, LA, pp. 344–351, 1996.
44. J. Park, V. Kanitkar, R. Uma, and A. Delis, “Optimal client clustering is NP-complete,” Technical Report, Polytechnic University, Brooklyn, NY, 1998.
45. R. Polamraju and W. Potter, “Databases for engineering applications,” in IEEE Proceedings of SOUTHEASTCON ’91, vol. 2. Williamsburg, VA, 1991.
46. M. Rabinovich, J. Chase, and S. Gadde, “Not all hits are created equal: Cooperative proxy caching over a wide-area network,” Computer Networks and ISDN Systems, vol. 30, nos. 22–23, pp. 2253–2259, 1998.
47. D. Saccà and G. Wiederhold, “Database partitioning in a cluster of processors,” ACM-Transaction on Database Systems, vol. 10, no. 1, pp. 29–56, 1985.
48. H. Sandhu and S. Zhou, “Cluster-based file replication in large-scale distributed systems,” in ACM SIGMETRICS and Performance ’92 Conference, 1992.
49. M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, “Coda: A highly available file system for a distributed workstation environment,” IEEE–Transactions on Computers, vol. 39, no. 4, 1990.
50. A. Sinha, “Client–server computing,” Communications of ACM, vol. 35, no. 7, 1992.
51. S. Su, H. Lam, S. Eddula, J. Arroyo, N. Prasad, and R. Zhuang, “OSAM*KBMS: An object-oriented knowledge base management system for supporting advanced applications,” in Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, 1993.
52. T.M.D. Team, “The miniRel relational DBMS,” University of Wisconsin, Madison, WI, 1989.
53. R. Tewari, M. Dahlin, H. Vin, and J. Kay, “Design considerations for distributed caching on the internet,” in Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Austin, TX, 1999.
54. S. Theodoridis and K. Koutroumbas, Pattern Recognition, Academic Press, London, 1999.

55. P. Triantafillou and C. Neilson, “Achieving strong consistency in a distributed file system,” IEEE Transactions on Software Engineering, vol. 3, no. 1, pp. 35–55, 1997.
56. M. Tsangaris and J. Naughton, “On the performance of object clustering techniques,” in Proceedings of 20th ACM SIGMOD Conference on the Management of Data, San Diego, CA, 1992.
57. Y. Wang and L. Rowe, “Cache consistency and concurrency control in a client/server DBMS architecture,” in Proceedings of the 1991 ACM SIGMOD Conference, Denver, CO, 1991.
58. V. Wietrzyk and M. Orgun, “Dynamic reorganization of object databases,” in Proceedings of the the 1999 IEEE International Database Engineering and Applications Symposium, Montreal, Canada, 1999.
59. K. Wilkinson and M. Neimat, “Maintaining consistency of client–cached data,” in Proceedings of the 16th International Conference on Very Large Data Bases, pp. 122–133, 1990.
60. C. Yu, C. Suen, K. Lam, and M. Siu, “Adaptive record clustering,” ACM Transactions on Database Systems, vol. 10, no. 2, pp. 180–204, 1985.
61. P. Yu, M. Chen, H. Heiss, and S. Lee, “On workload characterization of relational database environments,” IEEE Transaction of Software Engineering, vol. 18, no. 4, pp. 347–355, 1992.