Performance Evaluation of Centralized and Distributed Index Schemes for a Page Server OODBMS

Julie Basu
julie@cs.stanford.edu
Stanford University
Computer Science Department
and Oracle Corporation

Arthur M. Keller
ark@cs.stanford.edu
Stanford University
Computer Science Department
Stanford, CA 94305-9020, USA.

Meikel Pöss
poess@cs.tu-berlin.de
Technical University of Berlin
Computer Science Department
10587 Berlin, Germany

March 19, 1997

Abstract

Recent work on client-server *data-shipping* OODBs has demonstrated the usefulness of local data caching at client sites. However, none of the studies has investigated index-related performance issues in particular. References to index pages arise from associative queries and from updates on indexed attributes, often making indexes the most heavily used *hot spots* in a database. System performance is therefore quite sensitive to the index management scheme. This paper examines the effects of index caching, and investigates two schemes, one centralized and the other distributed, for index page management in a *page server* OODB. In the centralized scheme, index pages are not allowed to be cached at client sites; thus, communication with the central server is required for all index-based queries and index updates. The distributed index management scheme supports inter-transaction caching of index pages at client sites, and enforces a distributed index consistency control protocol similar to that of data pages. We study via simulation the performance of these two index management schemes under several different workloads and contention profiles, and identify scenarios where each of the two schemes performs better than the other.

1 Introduction

Object-oriented database systems (OODBs) have gained popularity in recent years, especially in the application areas of CAD/CAM and CASE. OODBs typically have client-server architectures, and commonly use the *data shipping* approach [6] to move relevant data from the central database server to local client caches. The data-shipping strategy allows query processing to occur locally at client sites, and exploits the resources of client workstations. A page (of 4K or 8K fixed byte

length) is usually the physical unit of data transfer between a client and a server, such systems being termed *page servers*. Page-level data shipping has been found to have superior performance under many different load conditions over *object servers*, which use individual objects as the *logical* units of data transfer [3].

Local caching and inter-transaction reuse of pages fetched in from the server can reduce network traffic between a client and the server, thus minimizing query response times. Despite the potential cost of maintaining the consistency of local caches, several studies of page server OODBs have demonstrated the general positive effects of client-side data caching on system performance [2, 3, 7, 8, 15, 16].

All of the above studies have examined in detail different techniques of data caching and consistency maintenance of data pages cached at client sites. However, a general assumption has been that transactions access the client cache *navigationally*, i.e., only through object IDs. For example, the simulation model adopted in [8] assumes an input transaction to be represented by a list of object references. In our view, this model is inadequate, in that it ignores the commonly-used *associative* queries and updates, which specify a target set of objects using general predicates on some attributes of an object class.

Supporting associative queries at client sites requires *content-based* access to cached data, i.e., access based on the values of the object attributes and not merely on their IDs. Indexes defined at the server are normally used during query planning and execution phases to provide efficient associative access to the central database. One way of providing associative access to a client cache is to allow caching of these index pages at the client site. However, the different factors involved in caching of central index pages, including the maintenance costs and effects on system performance, have not been investigated in earlier studies.

An important performance issue is that one index page generally contains many more entries compared to a data page, often making index pages the most heavily used *hot spots* in a database. System throughput is therefore sensitive to the number and frequency of index accesses and updates. Modification of any indexed attribute of a cached object will result in the corresponding index page(s) being updated. Previous studies of client-side data caching have not adequately modeled during analysis and simulation the costs arising from such index updates in the course of a transaction, even for navigational ID-based object writes.

This paper attempts to investigate the issues raised above, and to evaluate by simulation the performance of a page server OODB system supporting the usual navigational and also associative client cache access via index page caching. We extend the simulation model adopted in [3] to incorporate index-based associative queries in transactions, consider index read costs for associative

processing along with index update costs for both navigational and associative transactions, and quantify the benefits and drawbacks of index page caching at client sites. We consider a *distributed* index caching scheme in which inter-transaction caching of index pages is supported following a distributed consistency control protocol. This method of index management is contrasted against a *centralized* scheme that maintains all index pages at the server, with no client-side index caching being permitted within or across transactions. We develop simulators for both the schemes, and experimentally analyze the effects on system performance under different access patterns and contention profiles.

The rest of this paper is organized as follows. Section 2 reviews related work. In Section 3, we discuss the issues and trade-offs in index caching at client sites. Section 4 gives details of the system configuration, the simulation model, and the various cost parameters. Sections 5 and 6 describe the workloads considered and experiments that were performed, and present the simulation results. Finally, we summarize our conclusions in Section 7.

2 Related Work

As noted in the Introduction, several recent studies have established the performance benefits of client-side data caching schemes for OODBMSs [2, 3, 7, 8, 15, 16]. All of these papers have examined caching of pages in general, without considering index pages in particular. For example, in the simulation study of [3], some of the load profiles portray regions of high contention among clients. However, representing indexes simply as high contention data regions does not provide an adequate model of index behavior — as discussed in Section 3, access, update, and contention characteristics of index pages are very different from those of general data pages, and in particular from the data page read and write patterns considered in [3].

Among other related work, index partitioning among different sites in the context of distributed databases has been investigated in [12]. As in our work, range queries are a focus of their study also. However, in contrast to our dynamic caching environment, the partitioning in [12] is static and does not vary with query patterns at a site. Additionally, unlike our centralized client-server environment, query processing is distributed in nature, with possibly several sites computing partial query results. In our scenario, transactions are tied to their site of origin — i.e., multiple clients do not cooperatively work on a single transaction.

In a broader perspective, the issue of supporting associative access to a client-side cache has been addressed in [5] and [11] — the latter presents a *predicate-based* caching scheme while the former employs *ViewCache* techniques with local indexes on the client cache in the ADMS± system. A paper closely related to [11] is the *semantic* caching approach presented in [4]. Unlike these studies, using centrally defined indexes to support associative cache access is the subject of this paper.

3 Issues in Index Caching

We note the characteristics and use of index pages, and the benefits and costs of index caching. The discussion is with respect to a page server system supporting client-side caching.

3.1 Index Page Characteristics and Usage

Index pages differ from data pages in the following respects:

- **Index page entry size:** The size of an index page entry, conceptually consisting of an <attribute value, associated OID> pair, is normally much smaller than a data object size. Therefore, there are, on the average, many more entries per index page than there are objects in a data page, making index pages often the most heavily used *hot spots* in a database.
- **Index reads:** Index range reads always occur at the start of processing an associative range query, and may result in either clustered or unclustered data access. The number of data objects accessed per page, i.e., the *page locality* [6], may differ radically depending on whether the data is accessed via a clustered or an unclustered index. The simulation model of [3] does have page locality as a parameter, but no index lookup or update costs are considered.
- **Page reference pattern for index writes:** Modification of an indexed attribute in a data object will cause the corresponding index page(s) to be updated — the old index entry must be deleted, and a new index entry corresponding to the new attribute value must be inserted, possibly in a page different from that of the old entry. Zero or more index page updates may follow an object write. Thus, the index page referencing pattern is quite different from that of data pages.
- **Index write probability:** Previous simulation studies have generally investigated the effect of varying the data object write probability. The index write probability per data object update is a different quantity — it depends on whether the indexed attribute is modified. It should therefore be considered separately.

Consider, for example, a employee table or object class EMP(*name, title, salary, dept_id*) that records a name, title, salary and department for each employee. Assume there are two indexes defined on EMP: a clustered B+-tree index based on the foreign key attribute *dept_id*, and the other an unclustered B+-tree index on *salary*. Now suppose that the following associative query is submitted:

```
SELECT name, title, salary, dept_id FROM EMP
WHERE salary BETWEEN 50000 AND 70000
FOR UPDATE;
```

In order to process this query efficiently, a range scan of the *salary* index is necessary. Depending on the index management strategy, this reference to the index may happen either at the server or at the client, but in either case producing a list of qualifying object IDs. Since the index on *salary* is unclustered, EMP objects will be referenced randomly in the data pages. If the query predicate instead was "WHERE dept_id = 100," the set of EMP objects retrieved would be grouped closely together in successive data pages. The client processes the EMP object IDs one by one, fetching the corresponding data page from the server if it is missing from the local cache. If a fetched object is updated (based on some program logic), one or both of the indexes may need to be updated.

3.2 Costs and Benefits of Index Caching

The query optimizer at the server uses any available indexes to generate efficient execution plans for associative queries, by limiting the set of data pages that need to be examined. Indexes could also be used to provide associative access to a client cache. For any query that specifies target objects through a predicate involving an indexed attribute, the relevant index pages may be used to determine whether a client has all or some of the desired objects cached locally. If any queried object is locally available, then the corresponding data page need not be re-fetched from the remote server, thereby reducing network communication and improving query response times.

If the caching policy disallows storage and reuse of index pages at client sites, then all index page references for range queries and index updates at the client must be routed to the server, which may become a bottleneck in the system. Although inter-transaction caching of index pages can support local processing of associative queries, it requires the enforcement of a distributed index consistency control protocol, which may be expensive in certain update-intensive scenarios. Two particular index management schemes, one centralized and the other distributed, are described in the following section and are analyzed quantitatively in our simulation.

4 Centralized and Distributed Schemes for Index Management

We consider two different schemes, one centralized and the other distributed, for access and maintenance of the server index pages in a client-server environment. For both the schemes, we use the PS-AA caching method for data pages [3], which provides *adaptive* granularity for concurrency control and consistency maintenance of cached data, while using the page as the fixed unit of data transfer across the network. The PS-AA method of locking and replica control switches from page-based to object-based locking when finer-grained sharing is deemed better, and uses *callbacks* (first proposed in the context of the Andrew File System [10]) for coordinating updates. For a variety of workloads studied in [3], the adaptive page server following PS-AA caching strategy was found to have consistently good performance, generally outperforming the other static and dynamic caching strategies investigated.

4.1 Centralized Index Management

In this scheme, all index pages are centrally stored and managed exclusively by the server. They are not allowed to be cached by clients, and thus are treated very differently from data pages. A client submits each associative query to the server. The server searches the relevant index pages to determine which data objects fall within the query range, and responds with a list of qualifying object IDs. We assume that an object ID is in the common *structured* form, i.e., it contains a physical page number in its higher order bits and a logical slot number in the low order bits [1], so that the page in which an object resides is indicated by its ID. The client processes the received list of objects IDs sequentially, as in the case of the "next" operation on a cursor, and requests the server for the data page of an object missing from its local cache.

Any index entry delete and insert requests resulting from a data object update (for the old and new index entries respectively) are sent to the server for incorporating on the central index pages. We assume in this paper that propagation of index updates to the server is not *deferred*. The deferred approach is used by some commercial OODB systems to reduce network traffic, but may cause delayed detection of index errors.

The centralized index approach has the basic disadvantage that every access and update of index pages requires a round-trip communication with the remote server. The advantage is that central maintenance of the index at the server simplifies coordination of simultaneous updates to index pages by different clients. As shown in our simulation results, the performance of the centralized scheme in a client-server environment depends on the nature of the workload.

4.2 Distributed Index Caching

We now outline an inter-transaction index caching scheme which allows local storage of any index pages referenced (read or updated) by the client. Which index pages get cached locally depends on the data access and update pattern of the client. Whenever an index page has to be read for an associative query, the client requests the server for a copy of the index page only if it is not cached locally. Thus, cached index pages have implicit permission for local read.

Updating cached index pages requires coordination through the server. In a centralized scenario, we assume that an index page is latched at the server (e.g., via a semaphore) only for the duration of an actual insert or delete operation on it, and that advanced techniques such as index range locking or predicate/granular locks [9] are not used. For this case, the isolation model for concurrency control is that of cursor stability [9] — there is no protection against *phantoms*, and no *repeatable read* property for queries. This level of transaction consistency is popular in commercial database systems [9, 14], and is adopted in our study; our centralized and distributed index schemes both provide an isolation model comparable to cursor stability.

For index page writes, we follow a policy similar to the *Callback-Read* [7] scheme for data pages. Acquiring a write latch on an index page involves invalidating cached copies of the page at all clients other than the requestor, and granting the owner of the latch exclusive permission to update the page. The index page is sent to the requestor site if it is not already cached there. A client sends the index page in question back to the server immediately after the update, whereupon the distributed write latch is released, allowing any other clients waiting to read or write the same index page to proceed. The client may continue to cache the copy of the index page locally until it is flushed in response to an invalidation message from the server, or is aged out by the normal LRU buffer page replacement algorithm.

The main benefit of allowing index caching at client sites is that no communication is necessary with the server when index pages to be read are locally available at a client. However, client caching of indexes has the basic problem that update contention over shared index pages may increase network traffic and update costs, even when there is no sharing of data pages. Consider a situation where one half of an index page relates to objects cached at a client A, and the other half relates to a disjoint set of objects and pages cached at client B. Even if the update activity is relatively low at both clients A and B, substantial contention and loss of performance may occur due to the dependence of both clients on the shared index page. Our objective in this study is to quantify the performance characteristics of such a scheme.

5 Simulation Model

We now describe the details of our simulation model and the various system and cost parameters, focusing mainly on our refinements to explicitly model indexes. The overall client-server system architecture is represented in Figure 1.

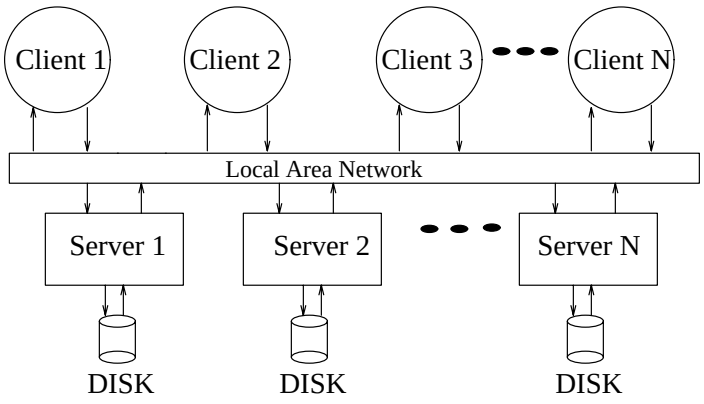


Figure 1. A typical client-server system communicating via a LAN.

5.1 Page-Server Simulation Model

Our simulation model for the page server is essentially the same as that of [3]. We have reproduced in Figure 2 below the basic page server simulation model from [3], with an additional "index manager" module on the server side for our index handling extensions. We describe here very briefly the general page server scheme — the details may be found in [3].

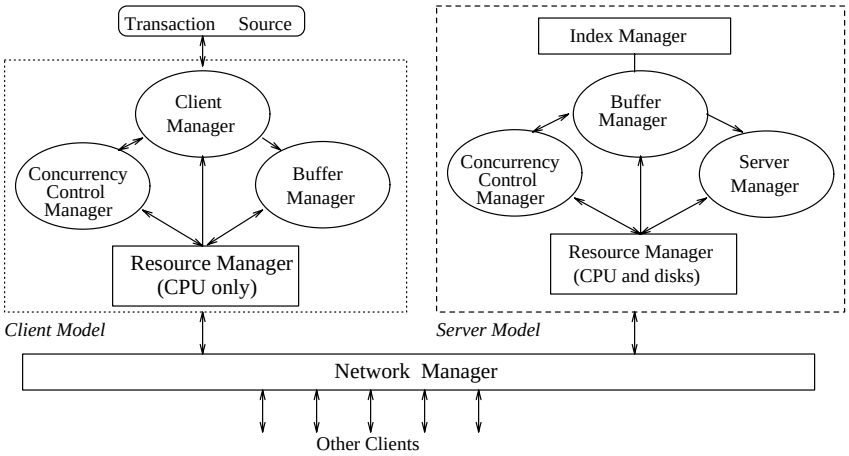


Figure 2. Simulation Model of a Page Server OODBMS

Pages (generally of fixed byte size) are the unit of data transfer and caching in the page server systems. Several studies, e.g., [6] and [7] have investigated the performance of page servers compared to object servers. This issue is not the focus of our work, and we assume page-based algorithms for data and index page transfer and caching. The unit of concurrency control may differ however from the data transfer and caching units, and based on the favorable results reported in [3], we adopt the PS-AA concurrency control method for data pages. However, for index page access and update, we experiment with the two page-based protocols, centralized and distributed, as defined earlier in Section 4. Like data pages, index pages are allowed to be cached in the distributed index scheme, but a concurrency control protocol different from the data pages is used, keeping in mind the special characteristics of index pages.

5.2 Modeling Indexes

We model each associative access as an index range read, the width of the "read window" depending on the parameter average transaction *size*, i.e., the average number of objects accessed by a transaction. More than one index page may be read if the range scan crosses index page boundaries. The start points of index scans are generated randomly, based on the index usage profile supplied for the particular client (workload profiles are described in detail in Section 5.4). The result of an index scan is a list of OIDs that is then processed by the client transaction, with object and index

writes occurring according to the specified workload.

For each index range read, the centralized scheme results in a round-trip to the server. Network and index lookup costs are associated with this operation. The distributed scheme incurs similar index read overhead in processing associative queries for which the index pages are not locally available. For large-sized transactions, the index read window may overlap two or more index pages. In keeping with the "one-page-at-a-time" functionality of the page server, the distributed scheme handles index page read (and also write) requests one index page at a time. Thus, unlike the centralized case, a single but large index range read may cause multiple round-trips to the server for the distributed scheme. This extra cost is reflected in our performance measurements, as discussed in the next section (Figure 3(b)).

Upon each data object write, one or more index updates may occur, based on the index write probabilities (parameters in our simulation model). For the centralized case, a round-trip to the server is required to update the appropriate indexes associated with the modified attribute value.[1] An index update request for the distributed scheme requires communication with the server to obtain exclusive update privileges for each index page updated, as described in Section 4.2. Once a write latch is granted on an index page, the index entries are updated (the old entry deleted and a new entry inserted) to reflect the data object write. All index writes are undone upon transaction aborts, which may be caused by the detection and resolution of deadlocks by the simulator. Since appropriate locks are held on the data objects themselves, updating the index before a transaction commits does not cause erroneous behavior — if a transaction reads an uncommitted index entry, it will subsequently block upon a read or write request for the associated data object until the updating transaction commits or aborts.

For the purposes of this study, we do not consider the effects of index page splitting and merging that may take place in a B-tree index or one of its variants. In other words, index page overflow and underflow is not considered. This assumption is not overly restrictive, since page splitting and merging are often relatively rare occurrences in practice, especially for light update loads. A result of this simplification is that only index pages at the leaf level that contain pointers to OIDs are considered pertinent for this study. All non-leaf index pages in a B-tree index can be assumed to be invariant for our simulation, and available read-only to all clients and the server. Availability of the non-leaf index pages allows a client to locally determine which leaf level index page must be accessed for an index scan or update.

Our simulator explicitly models two indexes, one clustered and the other unclustered, on the database. These indexes are named *cix* and *uix* respectively. It is important to consider both types of indexes, since the index usage, and data access and update patterns are different for the

[1] As in [3], data object modifications are not sent to the server until a transaction commits or aborts.

two. A query that uses a range predicate on an attribute with a clustered index will retrieve a set of objects grouped closely together in some set of data pages. On the other hand, access through an unclustered index will result in random data pages being fetched in. Clustered indexes are generally less likely to be updated, because data objects are generally clustered in pages according to an attribute that is infrequently updated, such as the department number of employee tuples in an employee relation.

During the initialization phase of the simulator, both types of index pages are explicitly populated with associated object IDs. Clustered index pages hold object IDs in the order they appear in data pages. That is, objects are placed in the clustered index pages sequentially according to their page IDs. Pages of the unclustered index are randomly populated with object IDs. To avoid aberrations in the results from a cold start with empty caches, both the server and the client caches are pre-loaded with pages. The server main memory buffer manager first loads in all index pages during the initialization phase, since it is likely that index pages will be frequently used by the clients. It fills up the remaining buffer space with data pages selected randomly. In contrast, each client loads into its cache a fraction of its hot data pages, and places cold data pages in the remaining space; no index pages are cached by the clients at the start of simulation.

5.3 System and Cost Parameters

General system and cost parameters and their values for the simulation experiments are listed in Table I – these are identical to those assumed in [3]. Additional index-related parameters and costs that are specific to our enhancements are defined separately in Table II. The simulator code is written in the Modula-2 based simulation language DeNet [13], and uses for the basic PS-AA page server the same code as in the simulator developed in [3]. Thus, we have the exact behavior for data page caching as reported in [3], with index page reads, writes and local index caching being our newly added functionality.

An index entry is assumed to be 16 bytes. Given our page size of 4K bytes, this implies that an index page contains 250 object entries, as opposed to 20 objects per data page. Our database contains in total 25,000 objects, and therefore the clustered and unclustered indexes occupy 100 pages each.

Index range reads incur a lookup cost, represented by *IxLookUpCost*, to locate from the root of the B+ index the leaf index page which holds the first object entry in the read interval. Once this index entry is located, each consecutive index entry is read, at a cost of *IxEntryReadCost*, until the end of the range read interval is reached. Index page updates also require a lookup, followed by acquiring a latch, with cost *IxLatchCost*, on each page updated. The actual update is assumed to have a cost of *IxPageUpdateCost*. As mentioned before, we do not consider index page splitting

Table I: General System and Cost parameters

Parameter	Meaning	Value
NumClients	Number of client workstations	10
ClientCPU	Instruction rate of client CPU	15 MIPS
ServerCPU	Instruction rate of server CPU	30 MIPS
ClientBufSize	Per-client buffer size	25% of DBsize
ServerBufSize	Server buffer size	50% of DBsize
ServerDisks	Number of disks at server	2 disks
MinDiskTime	Minimum disk access time	10 milliseconds
MaxDiskTime	Maximum disk access time	30 milliseconds
NetworkBandwidth	Speed of network communication	80 Mbits per second
PageSize	Size of a page (data transfer unit)	4096 (4K) bytes
DBsize	Size of the database	1250 pages (5 MB)
ObjsPerDataPage	Number of objects per data page	20 objects
FixedMsgCost	Fixed instructions per message	20,000 instructions
PerByteMsgCost	Additional instructions per message byte	10,000 per 4KB page
ControlMsgSize	Size of a control message	256 bytes
LockCost	Cost per lock/unlock pair	300 instructions
RegisterCopyCost	Cost to register/uregister a page copy	300 instructions
DiskCost	Cost of performing a disk I/O	5000 instructions
ReadObjCost	Mean cost to read an object	5000 instructions
WriteObjCost	Mean cost to write an object	10000 instructions
DataPageMergeCost	Cost to merge two copies of a data page	300 instructions per object

Table II: Index-related System and Cost Parameters

Parameter	Meaning	Value
NumClustIx	Number of clustered indexes on DB	1
NumUnclustIx	Number of unclustered indexes on DB	1
InitEntriesPerIxPage	Initial no. of entries per index page	250
NumCixPages	Number of clustered index (*cix*) pages	100
NumUixPages	Number of unclustered index (*uix*) pages	100
IxLatchCost	Cost per latch/unlatch of an index page	50 instructions
IxLookupCost	Cost to locate an index page entry given an object ID	1000 instructions
IxEntryReadCost	Cost to read the next entry in index range	10 instructions
IxPageUpdateCost	Cost to insert or delete an index page entry	2000 instructions

Table III(a): Workload Profile for Client i, i = 1..10.

Parameter	Meaning	Workload Type		
		HOTCOLD	**UNIFORM**	**HICON**
HotDataPgs	Hot range of data pages	h to $h+124$, $h = 125*(i-1)+1$	—	1 to 250
ColdDataPgs	Cold range of data pages	rest of DB	whole DB	rest of DB
HotCixPgs	Hot range of clustered index pages	c to $c+9$, $c = 10*(i-1)+1$	—	1 to 20
ColdCixPgs	Cold range of clustered index pages	rest of *cix* pgs	all *cix* pages	rest of *cix* pgs
HotUixPgs	Hot range of unclustered index pages	—	—	—
ColdUixPgs	Cold range of unclustered index pages	—	all *uix* pgs	—
AccHotDataProb	Probability of accessing a hot data page	0.8	—	0.8
AccHotCixProb	Probability of accessing a hot *cix* page	0.8	—	0.8
AccColdCixProb	Probability of accessing a cold *cix* page	0.2	uniform	0.2
AccColdUixProb	Probability of accessing a cold *uix* page	—	uniform	—

and merging costs. New entries are targeted randomly to the existing index pages, and we assume that all index pages handle entry insertion and deletion without overflow or underflow.

5.4 Workload Model

Transactions in our workload model are of two types: *associative* and *navigational*. The former type of transaction accesses data using an index; the latter is a list of data object ID references, as modeled in [3]. For the purposes of our simulation, an associative transaction consists of a single range query or update, which is expressed in terms of a linear range on either the clustered or the unclustered index. Processing of such a transaction commences by examining the necessary index pages, and by making a range scan over these pages to generate the list of object references. This list of object IDs is then processed one by one at the client site, fetching data pages as necessary from the server. An object write can trigger index updates, which result in write requests for index pages.

Client-specific workload profiles and general transaction parameters are summarized in Tables III(a) and III(b) respectively. The distribution of data and index pages among the different clients is described in Table III(a). These parameters vary by the choice of workload type — HOTCOLD, UNIFORM, or HICON, but are invariant for each client given any particular workload.

The HOTCOLD load profile considered for this study is similar to that of the HOTCOLD load studied in [3]. The HOTCOLD workload has a high degree of access locality per client and a moderate amount of data contention amongst the clients. As shown in Table III(b), each client has its own set of 125 hot data pages, access to which occurs with a probability of 80%. The hot page bounds of the clustered *cix* index matches the hot data page bounds for each client — each set of

10 *cix* pages corresponding to the hot data page range for the client is the hot *cix* page range for the client. The probability that a range read occurs in the hot *cix* page range is 80%. Associative access to data via range reads on the unclustered *uix* index is not considered for the HOTCOLD model, since it causes random page references. However, both the indexes are subject to updates upon individual object writes.

For the UNIFORM workload, data object references are assumed to be uniformly random over all the 1250 data pages, as in [3]. For this workload type, associative access to data via both clustered and unclustered indexes are considered. A range read starts randomly at any one of 100 pages of the accessed index, and then proceeds sequentially over the read window. Parameter settings for client i for the UNIFORM workload type are summarized in Table III(b).

The HICON workload is adapted from the corresponding workload in [3]. It is a skewed workload representing high data contention amongst the clients. As shown in Table III(b), all clients access the first 250 pages of the database, which are the shared hot data pages, with 80% probability. Only the clustered *cix* index is considered relevant for associative data access in this workload, the hot *cix* index range corresponding to the 250 hot data pages being the first 20 *cix* index pages.

Transaction characteristics for all workload types are listed in Table III(b). The set of first six parameters is kept fixed while the second set of parameters is varied in our experiments. The probability that a transaction is associative and accesses data via an index is represented by the *AssocProb* parameter. A value of 0 for *AssocProb* implies that none of the transactions use either of the two indexes for accessing data; although there are no index reads in this case, index writes may occur upon data object updates in navigational transactions. Given an associative transaction, *CixAccProb* denotes the probability that data is accessed via the clustered *cix* index. Only the UNIFORM workload supports associative data access via the unclustered *uix* index, and in this case, the corresponding parameter *UixAccProb* for the *uix* index is simply $(1 - CixAccProb)$.

An important characteristic of a transaction is its *size* or length, which is the number of data objects it accesses. In our model, there are four parameters that relate to the size of a transaction — *NumPages*, *PageLocality*, *CixReadSize*, and *UixReadSize*. The first two are applicable only for navigational transactions, while the latter two are relevant for associative transactions only. *NumPages* denotes the mean number of pages accessed by a navigational transaction. The size of a navigational transaction in terms of the number of data objects it accesses is determined by the *PageLocality* parameter. Based on the simulation parameter values adopted in [3], we have chosen for our experiments a (fixed) *PageLocality* range of 1-7 with a mean of 4. That is, a navigational transaction will access 4 data objects per page on the average, giving an overall transaction size of $(4 * NumPages)$ number of objects. In contrast, the size of an associative transaction is controlled by the *CixReadSize* and *UixReadSize* parameters. If data is accessed via the clustered index,

Table III(b): Transactions Parameters for All Workloads

Parameter	Meaning	Value
ThinkTime	Mean think time between transactions	0
PageLocality	No. of objects accessed per page by a navigational transaction	1-7 (min-max), mean 4
CixWrtProb	Probability of clustered index update per object write	0.1
UixWrtProb	Probability of unclustered index update per object write	0.8
Cix2pgWrtPb	Probability that a clustered index update modifies two index pages	0.8
Uix2PgWrtPb	Probability that an unclustered index update modifies two index pages	0.8
ReadOnlyProb	Probability that a transaction is read-only	varies
AssocProb	Probability that a transaction is associative	varies
CixAccProb	Probability that an associative transaction accesses data via the clustered index	varies
UixAccProb	Probability that an associative transaction accesses data via the unclustered index	$1 - CixAccProb$
NumPages	Mean no. of data pages accessed per navigational transaction	varies
CixReadSize	Mean no. of objects in a range scan of clustered index	varies
UixReadSize	Mean no. of objects in a range scan of unclustered index	varies
ObjWrtProb	Data Object Write Probability	varies

CixReadSize defines the mean number of index entries scanned by the range query. *UixReadSize* is the corresponding parameter for the unclustered index. Thus, *CixReadSize* and *UixReadSize* directly determine the length of an associative transaction in terms of data objects referenced.

A read-write transaction chooses to update a data object with probability *ObjWrtProb*. A data object write may result in updates to one or both the indexes, depending on the attributes updated. *CixWrtProb* and *UixWrtProb* respectively denote the probability of a clustered or an unclustered index update upon a data object write. These parameters have fixed values for our experiments, 10% and 80% respectively. Updating an index entry requires deleting the old entry for the old value and inserting a new entry with the new attribute value for the object. Whether the new entry is in a page different from the one holding the old entry determines whether one or two index pages are accessed for the index update operation. To accurately model this behavior, we use two additional parameters, *Cix2PgWrtPb* and *Uix2PgWrtPb*, to denote the probabilities that a new index entry is in a page different from the old one, when an index update occurs on the clustered or the unclustered index respectively. We do not consider object deletes and inserts in this study, although they could be incorporated in our simulation model.

Notice that there are two different parameters controlling object writes — namely, *ObjWrtProb* and *ReadOnlyProb*. The latter parameter controls object writes on a per-transaction granularity, while the former is a per-object write probability for a given read-write transaction. A read-only transaction occurs with probability *ReadOnlyProb*, and has no object writes. Therefore, the *ObjWrtProb* parameter is not relevant for a transaction that is read-only. Both object reads and updates are performed by a read-write transaction, which occurs with probability $(1-ReadOnlyProb)$, with *ObjWrtProb* being the probability of an object being written by it.

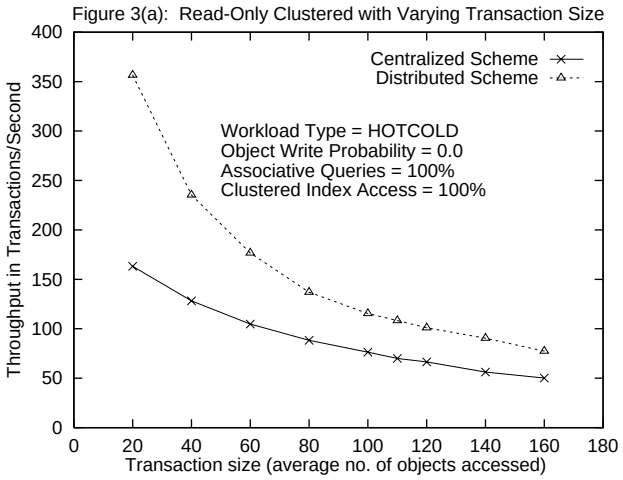
6 Simulation Experiments and Results

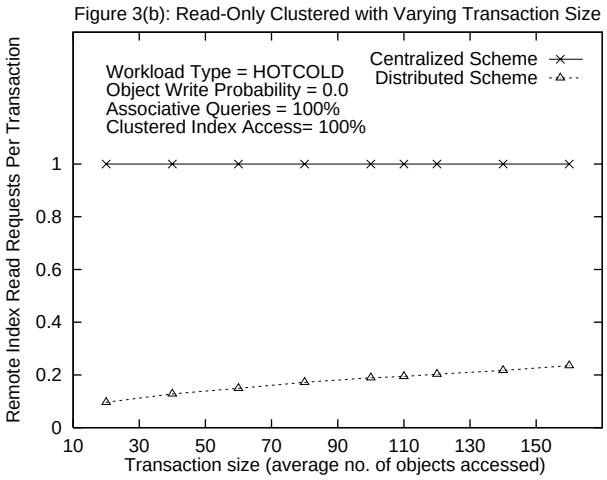
To check our implementation of the index management algorithms, we first verified that in the absence of associative queries (i.e., index reads) and index updates, both the centralized and distributed index algorithms yield exactly the same results. These results are also in agreement with the results reported in [3]. After this and a few other initial validation steps, several experiments were performed to evaluate the effects of the centralized and distributed index schemes under different load and access patterns. In the results reported below, we focus primarily on associative transactions and on index behavior. We use the throughput in terms of number of transactions per second (TPS) as our main measure of system performance. Our simulator keeps track of several other quantities, such as the number of remote index read and write requests. We use these two measures in particular to analyze the results of our experiments.

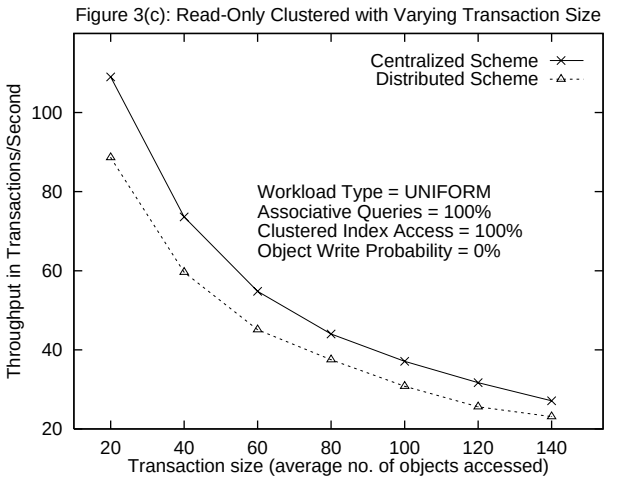
6.1 Read-Only Scenarios

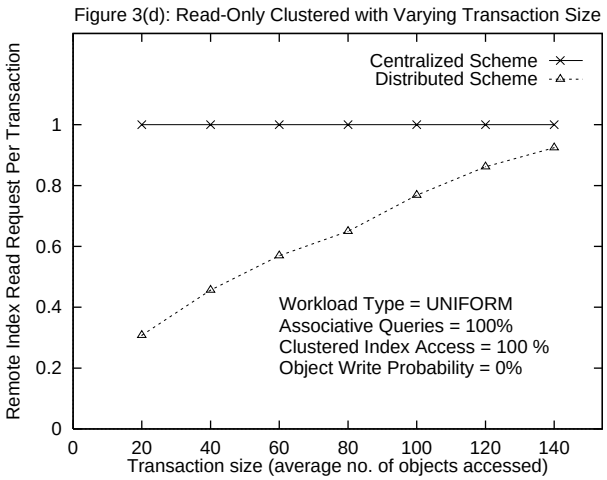
In order to explore the parameter space systematically, and to observe the effects of individual parameters, we first investigate the read-only behavior of the system with no object or index updates. In this scenario, there are two parameters that can be varied for all workloads: (i) the transaction size (controlled by $CixReadSize$ and $UixReadSize$ for an associative transaction, and by $NumPages$ for a navigational one), and (ii) $AssocProb$, the probability of a transaction being associative. Additionally, for the UNIFORM workload only, the $CixAccProb$ parameter can be varied to control the ratio of data access via the clustered versus the unclustered index.

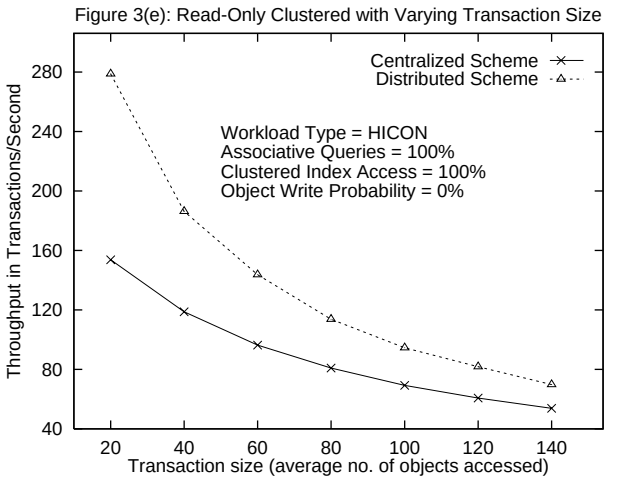
Figures 3(a) through 3(e) give the results of varying the transaction size for all three workloads, keeping the parameters $AssocProb$ and $CixAccProb$ constant at values of 1.0 and 1.0 respectively. The scenario in Figure 3(f) is similar, except that it uses the unclustered index for data access in a UNIFORM workload with $CixAccProb$ being 0.0.











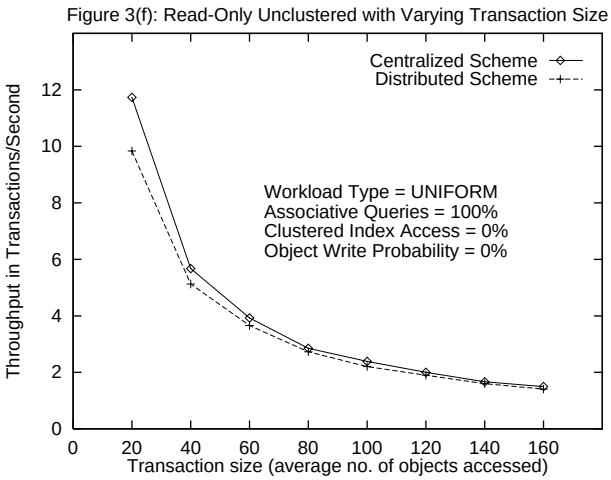


Figure 3(a) shows the effect of varying transaction size on the system throughput for the HOT-

COLD workload when there are no object writes, all transactions are associative, and all range reads are on the clustered index. It can be seen from the graph that the distributed scheme does much better than the centralized one over the range read sizes considered. The reason for this behavior is explained by Figure 3(b), which plots the number of index read requests sent to the server for the same load profile as in Figure 3(a). For the centralized scheme, each transaction causes exactly one remote index read request irrespective of the transaction size. In contrast, the distributed scheme fetches an index page from the server only if there is a cache miss for the page, and therefore the average number of requests for an index page is less than 0.25 over a wide range of transaction sizes. This number is 0.1 for a transaction size of 20 objects, and rises slowly to about .22 for a transaction size of 160 objects.

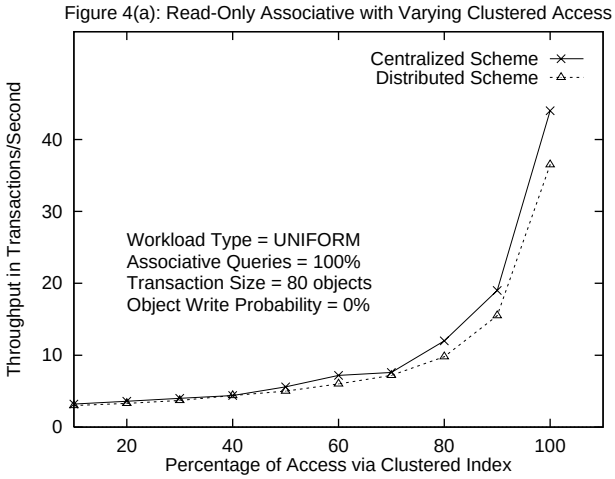
The reason for the slight upwards slope in the curve for the distributed scheme in Figure 3(b) is twofold. Firstly, recall that our simulator fetches index pages one at a time for range reads when the read interval overlaps two or more index pages. As the average index read size increases, more associative transactions reference two index pages instead of one at the start of processing, requiring two separate trips to the server. This penalty is not paid by the centralized scheme, which fetches a list of qualifying OIDs from the server in one round-trip. Secondly, increasing transaction sizes result in a larger number of data pages to be cached per transaction, causing some index pages to be aged out of the cache and decreasing the cache hit ratio for index pages.

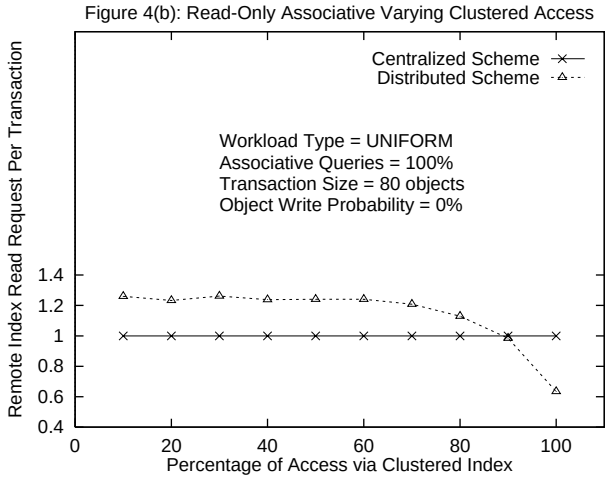
Figures 3(c) and (d) are the counterparts of Figures 3(a) and 3(b) for the UNIFORM workload. In contrast with the HOTCOLD results, the distributed scheme actually performs *worse* than the centralized in the UNIFORM case, converging to about the same performance for large transaction sizes. This reversal of performance happens despite the fact that the number of index read requests sent to the server is less for the distributed scheme compared to the centralized one (Figure 3(d)), just as in the HOTCOLD case (Figure 3(b)). The reason for this behavior of the distributed scheme is that random access to all 100 clustered index pages causes caching of more index pages for the UNIFORM case compared to the HOTCOLD one, which effectively reduces the space available for data pages in the client cache, thereby causing data page misses and decreasing the throughput.

The behavior of the HICON workload for the scenario corresponding to Figure 3(a) is shown in Figure 3(e). As in the HOTCOLD case, and in contrast with the UNIFORM one, the distributed scheme out-performs the centralized one for all transaction sizes. However, a comparison of Figures 3(a) and 3(e) shows that the throughput of the distributed scheme for any transaction size in the HICON case is lower than the corresponding value for HOTCOLD. This behavior is expected, since 20 index pages are hot in this workload compared to 10 in the HOTCOLD case, effectively causing more index caching and reducing the client buffer size by 10 pages.

Of our three workloads, only the UNIFORM workload involves associative data access via the unclustered index. Figure 3(f) shows the system throughput with varying transaction size in the

UNIFORM workload using the same parameter settings as in 3(c), except that $CixAccProb$ is set to 0.0; that is, for Figure 3(f), all associative queries use the unclustered index. The throughput for the distributed scheme slightly trails that of the centralized one over the entire range of transaction sizes. Notice that the throughput is much lower compared to the clustered access case of Figure 3(c), and that the relative difference in performance of the two index schemes is not as large. The reason for this behavior is that random data page references due to unclustered access reduce the performance of both the centralized and distributed schemes, and cause nearly equal number of data page misses in both cases.





Next, consider the read-only UNIFORM case with varying $CixAccProb$, i.e., varying access via the clustered index for associative queries. Figures 4(a) and 4(b) show the throughput and the index read requests in this scenario for a transaction size of 80 objects. The throughput of the distributed scheme lags slightly behind the centralized one over the entire range of clustered access, with performance of both schemes increasing rapidly beyond the point where 80% of associative access is through the clustered index. As shown in Figure 4(b), index page hits in the distributed scheme also increase for 80% and greater values of clustered access, with remote index read requests per transaction falling beyond the constant 1 for the centralized scheme. However, there is no cross-over in the throughput curves, because uniform access of the 200 *cix* and *uix* index pages and their caching reduces the effective client buffer size in the distributed scheme.

The last read-only scenario for all workloads consists of varying the parameter $AssocProb$ to control the percentage of associative queries. The results are presented in graphs 5(a) through 5(h). The transaction size is kept constant at 80 objects, and associative data access is entirely through the clustered index for the cases in Figures 5(a) through 5(e). Figures 5(g) and 5(h) are for the UNIFORM workload, and are similar to 5(c) and 5(d) except that data access occurs via the unclustered index.

Figure 5(a): Read-Only Clustered with Varying Associative Access

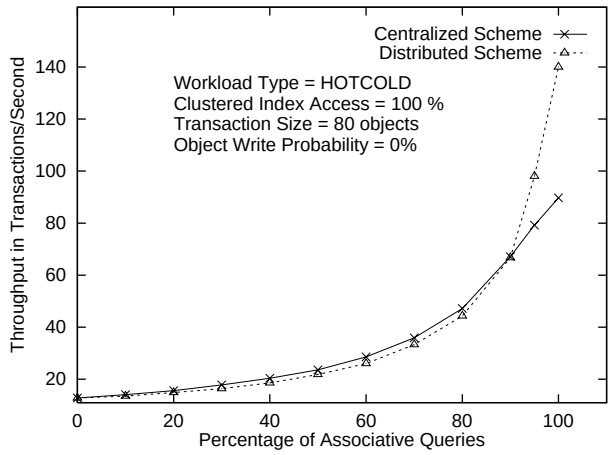


Figure 5(b): Read-Only Clustered with Varying Associative Access

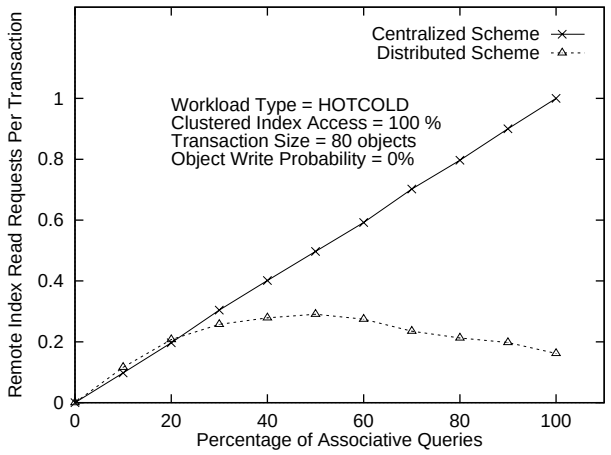


Figure 5(c): Read-Only Clustered with Varying Associative Access

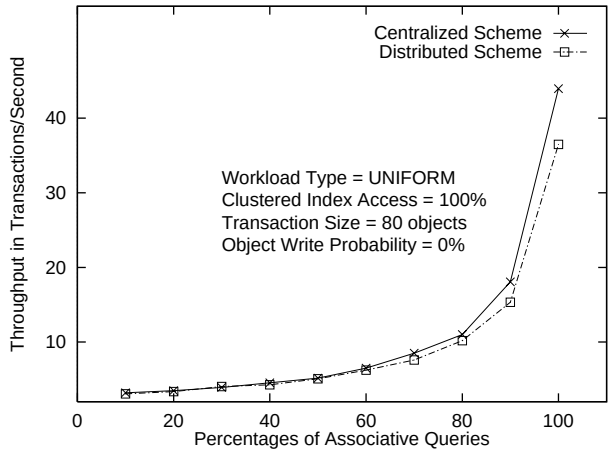


Figure 5(d): Read-Only Clustered with Varying Associative Access

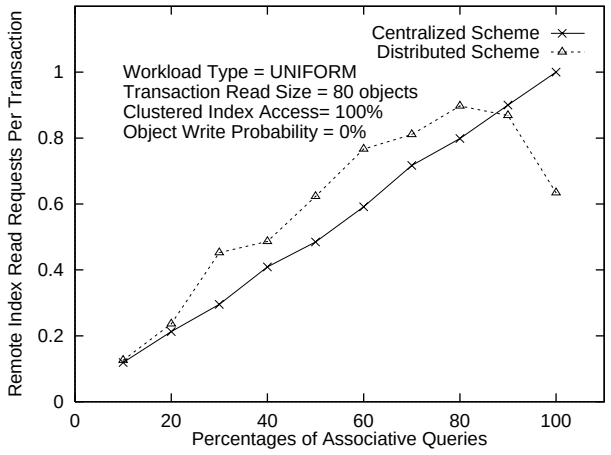


Figure 5(e): Read-Only Clustered with Varying Associative Access

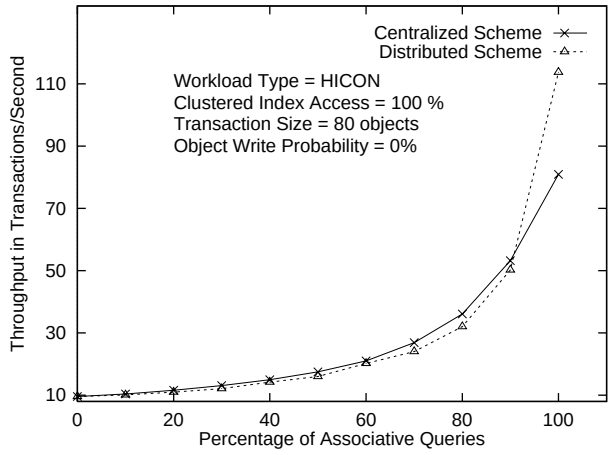
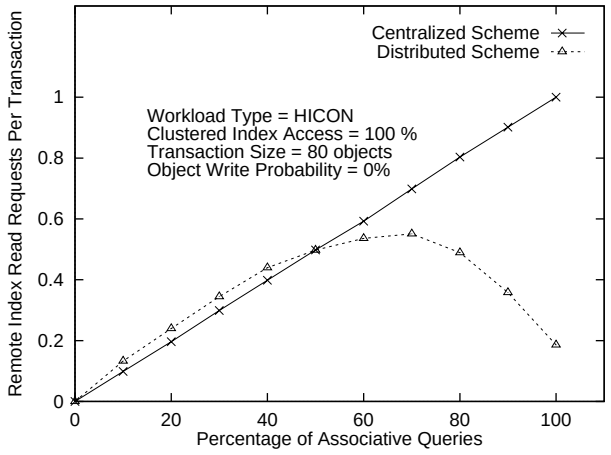


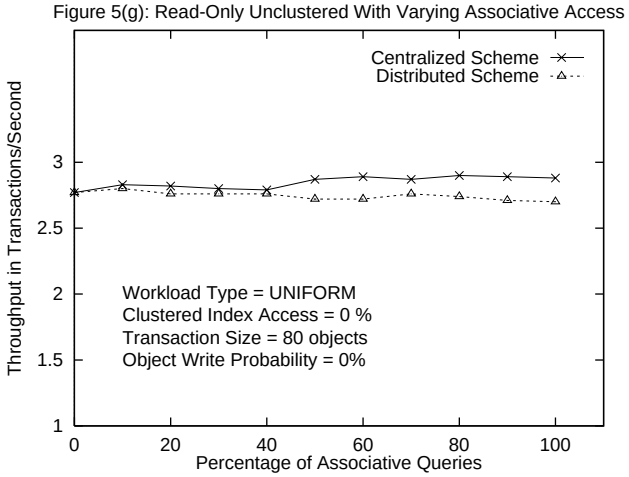
Figure 5(f): Read-Only Clustered with Varying Associative Access

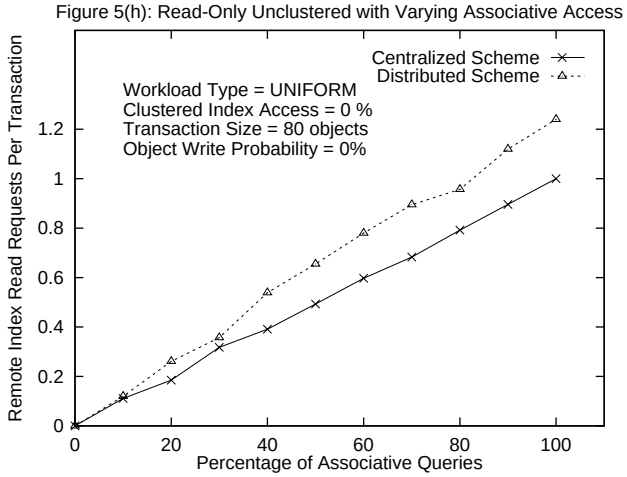


As shown in the above figures, varying the ratio of navigational versus associative access through the clustered index causes the throughput to vary substantially for all the three workloads. The

distributed scheme starts out as slightly worse than the centralized in the HOTCOLD workload (Figure 5(a)), but does better beyond 85% associative access. Similar behavior is observed for the HICON workload, with the throughput cross-over occurring slightly higher at 90% associative access. For both of these workloads, medium and high values of associative clustered access generate a lower number of index page read requests at the server for the distributed scheme than in the centralized (Figures 5(b), 5(d), and and 5(f)), indicating that index page caching is indeed effective in increasing the throughput by causing index hits for associative queries.

In contrast, for the UNIFORM workload, Figure 5(c) does not show a cross-over in throughput for the two index schemes. The number of remote index read requests as plotted in Figure 3(d) is higher for the distributed scheme until about 85% associative access, when it falls below that of the centralized. However, because of the uniform access to a larger number of index pages compared to the HOTCOLD and HICON workloads, and the resulting decrease in cache space for data pages, the throughput for the distributed scheme in the UNIFORM case does not rise enough to beat the centralized even for 100% associative data access, although the performance of the two schemes are comparable over the entire range of associative access.





Figures 5(g) and 5(h) show the case of unclustered access for varying percentage of associative queries in the UNIFORM workload. Due to the random nature of data page references with unclustered access, the performance of the both the index schemes deteriorates compared to the clustered case of Figure 5(c), and remains more or less constant with varying associative access. The distributed scheme causes slightly higher index read requests than the centralized, and unlike Figure 5(d), does not fall off for high ratios of associative access. This behavior can be attributed to that fact that uniform access to the unclustered index causes a larger number of random index and data page references compared to the clustered, reducing cache hits and the reuse of index and data pages at the client site.

6.2 Read-Write Cases

We have performed several sets of experiments varying the two major write parameters in our model, namely, *ReadOnlyProb* and *ObjWrtProb*. All of our results cannot be reported in this paper due to space constraints, so we discuss below some representative cases. The general trend we observed was that in the presence of update (read/write) transactions, there is significant deterioration in performance for the distributed index scheme, even with quite low object write probabilities. This behavior is not surprising given the cost associated with acquiring a distributed latch on an index page. However, as demonstrated in Figures 6(a) through 6(d), the ratio of update versus read-only transactions and the workload type are important factors in determining whether the centralized index scheme performs better than the distributed or vice-versa.

Figure 6(a): Read-Write Clustered With Varying Update Transactions

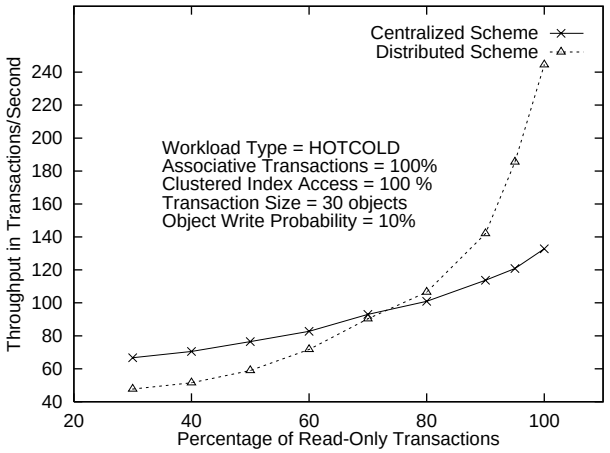
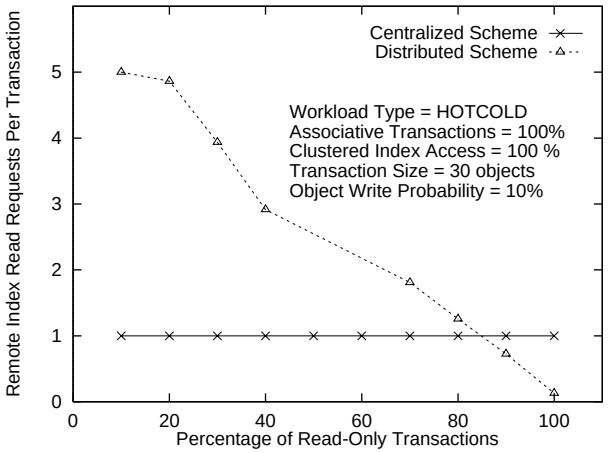


Figure 6(b): Read-Write Clustered With Varying Update Transactions



Figures 6(c): Read-Write Clustered with Varying Update Transactions

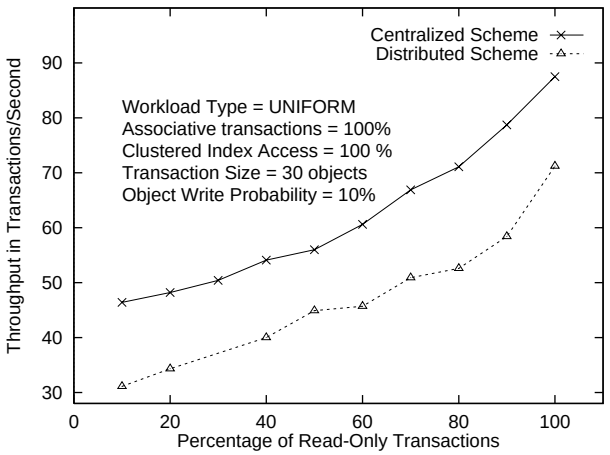
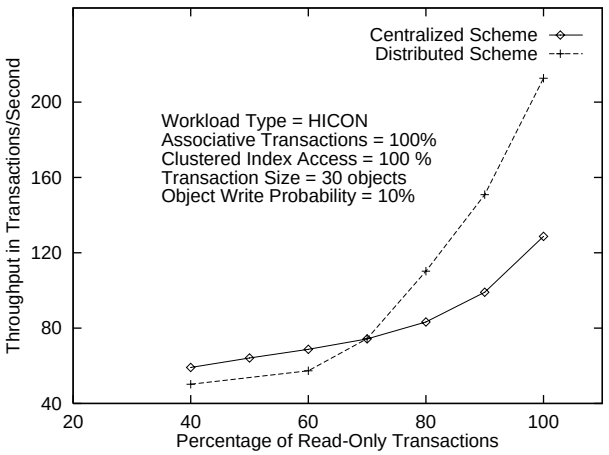


Figure 6(d): Read-Write Clustered with Varying Update Transactions



Figures 6(a) and 6(b) show the effect of varying the ratio of read-only versus update transactions for the HOTCOLD workload, using associative transactions of size 30 objects and the clustered index. The object write probability is 10%. The cross-over in throughput occurs when about 75% of the transactions are read-only, beyond which point the performance of the distributed index

scheme increases rapidly, exceeding the much slower growth for the centralized. The number of remote index read requests per transaction, as plotted in Figure 6(b), shows a rapid decrease for the distributed scheme with increasing number of read-only transactions, falling below that of the centralized when update transactions are less than 15% of the transaction load. Similar behavior is observed in the case of the HICON workload, with the cross-over in throughput occurring at a mix of 70% read-only queries and 30% update transactions (Figure 6(d)).

The performance characteristics of the UNIFORM workload under the same conditions are quite different from the HOTCOLD and HICON cases, as illustrated in Figure 6(c). In keeping with the behavior observed in the read-only scenarios of Section 6.1, the distributed index scheme never beats the centralized one, even with a 100% read-only load. Uniformly random access to all the 100 pages of the *cix* index causes a larger number of index pages to be cached in the client buffer, reducing the data page hits and the throughput.

Figure 7(a): Read-Write Clustered with Varying Object Writes

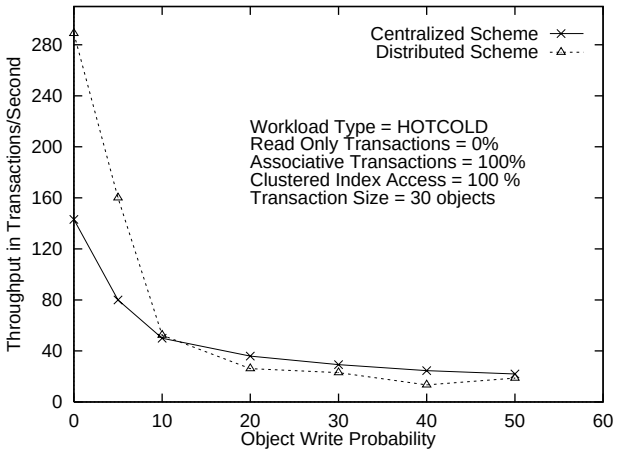


Figure 7(b): Read-Write Clustered with Varying Object Writes

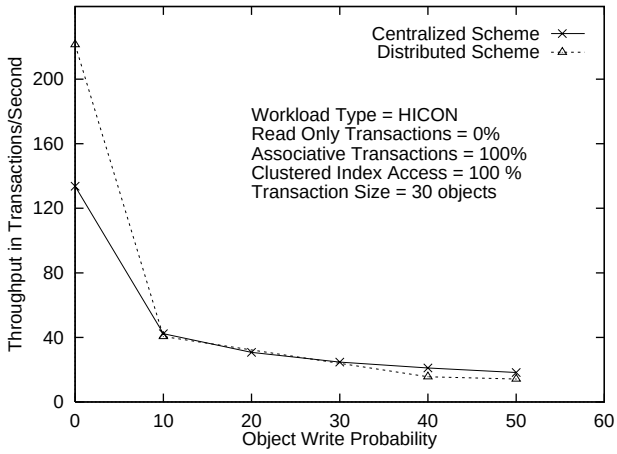


Figure 7(c): Read-Write Navigational with Varying Object Writes

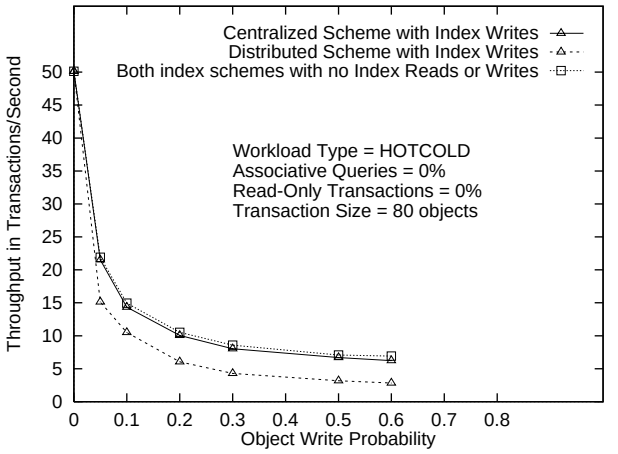
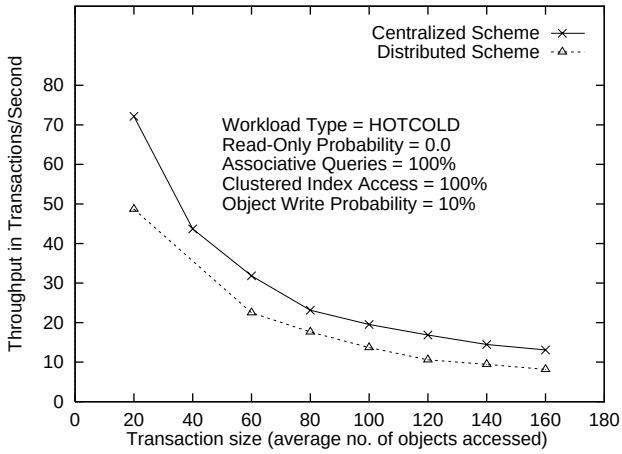


Figure 7(d): Read-Write Clustered with Varying Transaction Size



Figures 7(a) and 7(b) report the throughput of the HOTCOLD and HICON workloads with varying probability of object writes. For low values (less than 10%) of object update probability, the distributed index performs better than the centralized one with the given parameter settings. The performance of the UNIFORM workload (not shown here due to space restrictions) is different in that the centralized scheme is better than the distributed one for all object write probabilities.

Figure 7(c) above shows the system performance in terms of throughput for a load of exclusively navigational queries and mean transaction size of 80 objects. Notice that the centralized scheme closely follows the curve of no index reads or write. However, the distributed scheme with the same parameters lags behind, even for small values (5%) of the data object write probability.

Figure 7(d) is the read-write counterpart for the read-only scenario of Figure 3(a). The object write probability is only 10%, but all transactions are read-write. The performance of the distributed index scheme trails the centralized one for the whole range of transaction sizes from 20 through 160 objects.

6.3 Mixed Load Behavior

Our simulation model allows for associative and navigational transactions, clustered and non-clustered access, read-only versus read-write transactions, as well as client-specific definition of data and index usage. We have explored "mixed" loads where the above parameters have intermediate values in their allowable ranges. Figures 8(a) and 8(b) show a couple of our experimental results.

Figure 8(a): Read-Only Mixed Load with Varying Associative Access

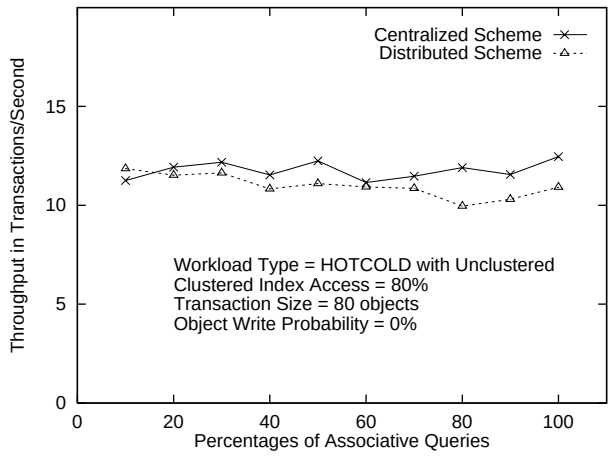
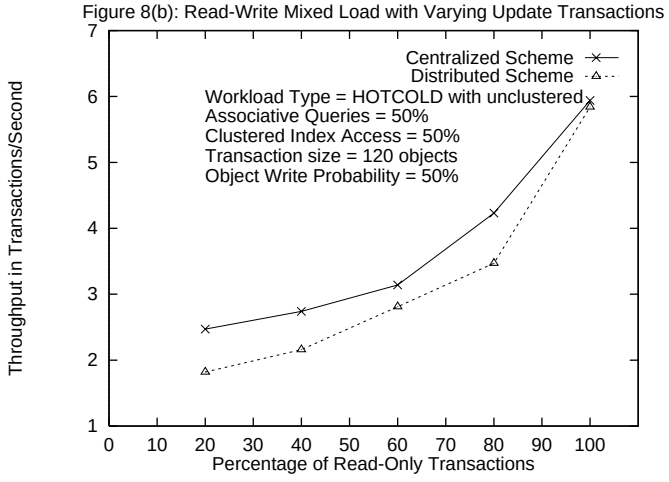


Figure 8(b): Read-Write Mixed Load with Varying Update Transactions



For Figure 8(a), we allowed a small percentage (20%) of unclustered data access in the HOTCOLD workload. This reduces the data skew to a more uniform usage. As in the read-only UNIFORM case, the centralized and distributed schemes have comparable performance for all ratios of associative versus navigational queries. Figure 8(b) is a Read-Write mixed load scenario,

with 50% associative queries, 50% clustered access and a 50% object write probability for transactions of size 120 objects. The performance of the centralized and distributed scheme are both quite poor, with the difference between the two decreasing with larger number of read-only transactions.

7 Conclusion

Indexes can provide efficient associative access to data, and their access and maintenance are important performance considerations for a database system. Previous studies in client-side caching for page server OODBs have not specifically considered index update costs and index-based queries, or index page caching in particular. We have extended the page server simulation model of [3] to incorporate clustered and unclustered indexes and associative (range) queries, and have evaluated through detailed simulation experiments the performance of two different schemes, one centralized and the other distributed, for index management in a page server OODB. The former executes all index read and write operations at the server site, while the latter allows index page caching and reuse at client sites following a distributed consistency control protocol. We have considered three separate workload types, HOTCOLD, UNIFORM, and HICON, modeling different data and index usage patterns at the clients, and have analyzed performance of the two index schemes under various parameter settings.

Reviewing our results, we find that for the UNIFORM workload, the centralized index always performs better than or is comparable to the distributed scheme in read-only as well as read-write scenarios. This result, although rather surprising for the read-only case, is due to the fact that index access is uniform in this workload, leading to a larger number of references to different index pages and their caching in the client buffer. The effective buffer space for data pages at the client site is thus reduced, and so is the system throughput.

The HOTCOLD and HICON workloads represent non-uniform data access by the clients, with contention characteristics being different for the two. For read-only scenarios in both of these workloads, distributed indexes offer substantial performance benefit over the centralized scheme when a large portion of associative access (85% or more) is through the clustered index. In other read-only cases, centralized indexes are competitive with distributed indexes. However, as the transaction size is decreased, the distributed scheme makes larger performance gains than the centralized one due to effective local reuse of index pages.

The picture for HOTCOLD and HICON cases is quite different in the presence of many updates; centralized indexes in general perform better than distributed, even for rather small write probability (10%) of data objects affecting an index. This behavior is a result of the high cost involved in obtaining exclusive distributed write latches on index pages. The gap narrows, however, as the percentage of read-only transactions increases. Cross-over points in throughput were

Table IV: Index Performance with Different Workloads, Transaction size 30 objects

Workload Type	Read-Only Transactions	Read-Write Transactions with 10% Object Write Probability	
		< 75% Read-Only	> 75% Read-Only
UNIFORM, > 70% clustered, all associative	Centralized better	Centralized better	Centralized better
UNIFORM, < 70% clustered, all associative	Both schemes comparable	Centralized better	Centralized better
HOTCOLD/HICON, > 90% associative, all clustered	Distributed better	Centralized better	Distributed better
HOTCOLD/HICON, < 90% associative, all clustered	Both schemes comparable	Centralized better	Both schemes comparable

obtained in our experiments with a varying mixture of read-only and read-write transactions, with the distributed scheme overtaking the centralized for high occurrences (about 70% or more) of read-only queries. We suggest that centralized indexes be used whenever there are a significant percentage of transactions updating indexes; otherwise, distributed index are better, at the cost of some architectural and implementation complexity.

We summarize in Table IV the behavior of the centralized and index schemes for different workload types. The results hold for a transaction size of 30 objects and an object write probability of 10%. The behavior is quite similar for larger transaction sizes; however, increasing the object write probability reduces the performance of the distributed scheme.

In conclusion, indexes behave very different than data in a page server OODBMS. While client-based query processing appears profitable for data pages, centralized (i.e., server-based) index processing seems superior for some workloads. In light of these results, we believe a "hybrid" approach of both server and client-based query processing might be the best alternative for a page server OODB. Further work remains to be done in this area to clearly identify the performance trade-offs, taking into account both data and index usage patterns. Advanced techniques like index range locking, or using the PS-AA locking and replica control protocol for both data and index pages are some topics for future work.

Acknowledgements

We are very grateful to Prof. Michael Carey for providing us the source code of the simulator developed for the work reported in [3], and to Markos Zaharioudakis for clarifying several questions on the implementation of the simulator. Their help has greatly simplified our simulation task.

The work of Julie Basu was supported in part by Oracle Corporation, and by an equipment grant from Digital Equipment Corporation.

References

[1] R.G.G. Cattell, *Object Data Management*, Addison Wesley, Reading, MA, 1991.

[2] M. Carey, M. Franklin, M. Livny, and E. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architecture," *ACM SIGMOD Intl. Conf. on Management of Data*, Denver, CO, May 1991, pp. 357-366.

[3] M. Carey, M.J. Franklin, and M. Zaharioudakis, "Fine-Grained Sharing in a Page Server OODBMS," *ACM SIGMOD Intl. Conf. on Management of Data*, Minneapolis, MI, May 1994, pp. 359-370.

[4] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan, "Semantic Data Caching and Replacement," *22nd Intl. Conference on Very Large Data Bases (VLDB 96)*, Bombay, India, September, 1996, to appear.

[5] A. Delis and N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures," *18th Intl. Conf. on Very Large Data Bases*, Vancouver, B.C., Canada, 1992, pp. 610-623.

[6] D. J. DeWitt, D. Maier, P. Futtersack, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," *16th Intl. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990, pp. 107-121.

[7] M.J. Franklin, "Caching and Memory Management in Client-Server Database Systems," *PhD thesis*, University of Wisconsin-Madison, 1993.

[8] M.J. Franklin, M.J. Carey, and M. Livny, "Local Disk Caching for Client-Server Database Systems," *19th Intl. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993, pp. 641-654.

[9] J. Gray and A. Reuter, "Isolation Concepts," in *Transaction Processing: Concepts and Techniques*, San Mateo, CA, Morgan Kaufmann Publishers, 1993, pp. 403-406.

[10] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems,* Vol. 6, No. 1, February 1988, pp. 51-81.

[11] A.M. Keller and J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures," *The VLDB Journal,* Vol. 5, No. 1, Jan 1996, pp. 35-47.

[12] J. Liebeherr, E.R. Omiecinski, and I.F. Akyildiz, "The effect of index partitioning schemes on the performance of distributed query processing," *IEEE Transactions on Knowledge and Data Engineering*, June 1993, vol.5, no.3, pp. 510-522.

[13] M. Livny, "DeNet User's Guide (Version 1.5)," Computer Sciences Department, University of Wisconsin-Madison, 1990.

[14] Oracle Corporation, "Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7," White Paper, Part No. A33745, July 1995.

[15] Y. Wang and L.A. Rowe, "Cache Consistency and Concurrency Control in a Client-Server DBMS Architecture," *ACM SIGMOD Intl. Conf. on Management of Data*, Denver, CO, May 1991, pp. 367-376.

[16] K. Wilkinson and M.-A. Neimat, "Maintaining Consistency of Client-Cached Data," *16th Intl. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990, pp. 122-133.