Centralized Versus Distributed Index Management in a Page Server OODBMS

Julie Basu
Stanford University
and
Oracle Corporation

Arthur M. Keller *
Stanford University
Computer Science Department
Palo Alto, CA 94305-2140

June 10, 1996

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 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 behavior 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.

*For more information please write to ark@cs.stanford.edu.

1 Introduction

Client-server configuration is currently the most popular architecture for database systems. In such a system, a central repository of data is managed by one or more servers while transactions are initiated and run at client sites. The client interact with the server(s) via a local area network. A server is typically a mainframe or a minicomputer, with clients being independent workstations. A traditional assumption has been that resources of clients are quite limited. Accordingly, the role of clients is often restricted to transmission of SQL queries across the network to the server, and presentation of the received results to the user. However, with the continuing growth in performance of workstations, the validity of this assumption comes into question. It is increasingly common to find clients that have substantial local computing power and storage capacity. Such clients are capable of performing intensive computations on their own, using the database as a remote resource that is to be accessed only when necessary. Increased client functionality and autonomy can potentially relieve the central servers of congestion and overloading, and lead to better overall system performance and scalability.

Object-oriented database systems (OODBs) have gained much 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 [5] to move relevant data from the central database 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 reuse of pages fetched in from the server can reduce network traffic and communication overhead between a client and the server, thus minimizing query response times. Despite the potential cost of maintaining 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, 6, 7, 13, 14].

All of the above studies have examined in detail different techniques of data caching and cache consistency maintenance of data pages at client sites. However, a general assumption has been that transactions access the client cache only through object or page IDs. For example, the simulation model adopted in [7] assumes input transactions to be represented by a string of object references. In our view, this model does not reflect reality, because *associative* queries and updates, which often form a significant part of real-life transactions, are ignored.

An associative query specifies a target set of objects using general predicates on some attributes of an object class. Effective client cache reuse for such queries requires associative access to cached data, i.e., access based on the values of object attributes and not merely on their IDs. Centrally defined indexes are commonly used during query planning and execution phases to provide efficient associative access to the central database. Caching these index pages *at client sites* can also improve reuse of local caches. However, the different factors involved in index caching, including its effects on system performance, have not been investigated in detail in earlier studies.

Another 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 performance is therefore sensitive to the number and frequency of index accesses and updates. Modification of any indexed attribute of a cached object will cause the corresponding index page(s) to be updated, resulting in a round-trip to the server. 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 non-associative ID-based object writes.

This paper attempts to provide answers to the questions raised above. We extend the simulation model adopted in [3] to incorporate index access and update costs, consider index-based associative queries in transactions, and investigate possible benefits and drawbacks of index caching at client sites in a page server OODBMS. We consider a *distributed* index caching scheme in which inter-transaction caching of index pages is allowed 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. The centralized scheme therefore requires communication with the server for all (read and write) references to index pages. 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, simulation model, and cost parameters. Sections 5 and 6 describe the workloads and experiments that were performed, and present the simulation results. Finally, we summarize our conclusions, outline work currently in progress, and preview future work 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, 6, 7, 13, 14]. All of these works have examined caching of pages in general, without considering index pages in particular. Consider, for example, the simulation model adopted in the study of [3]. Some of the load profiles investigated in this work portray regions of high contention among clients. One may suggest that index behavior could be investigated in such a system model by representing them as high contention data regions. We believe this approach provides an inadequate model of indexes, since access, update, and contention characteristics of index pages are very different from those of data pages. The data access and update patterns studied in [3] do not represent realistic index read and write patterns.

Among other related work, index partitioning among different sites in the context of distributed databases has been investigated in [11]. In contrast to our dynamic caching environment, the partitioning is static and does not vary with query patterns at a site. Additionally, unlike our centralized client-server environment, the 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. Their simulation experiments are however quite pertinent to the work reported in this paper, since range queries are a focus of their study also.

In a broader perspective, the issue of supporting associative access to a client-side cache has been addressed in [10] and [4] — the former presents a *predicate-based* caching scheme while the latter employs *ViewCache* techniques on the client cache in the ADMS± system. These schemes are alternative approaches to using centrally defined indexes for associative query execution, which is the focus of our investigation in this paper.

3 Issues in Index Caching

In this section, we discuss the characteristics and use of index pages, and qualitatively analyze the benefits and costs of index caching. We also compare and contrast the nature of index page usage against that of data pages. The discussion is with respect to a page server system supporting client-side data caching.

3.1 Index Access and Update

Indexes define associations of objects to attribute values, that is, they organize data object IDs into groups related by a common attribute value. Such indexes (or access paths) are frequently defined on the central database at the server and are used by the query optimizer at the server site to generate efficient query execution plans whenever the set of objects to be retrieved for a query is constrained on the indexed attribute(s). Access through an index generally limits the set of data pages that need to be examined, thus speeding up query execution. The cost of this speed-up is that the index has to be maintained for every update operation on the indexed attribute, i.e., for every insert, delete, or update of a row or object in the table or object class.

Observe that index pages 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,[1] thereby reducing network communication and improving query response times.

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 a 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 index pages on the *salary* attribute 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 accessed index is unclustered, EMP objects will be referenced randomly in the query range. The client processes these EMP objects one by one, fetching them in from the server if there are missing from the local cache. If a fetched object is updated (based on some program logic), one or both of the two indexes may need to be updated.

3.2 Special Characteristics of Index Pages

To appreciate why index page caching must be considered separately, consider the following characteristics of index pages that are substantially different from data pages:

- **Index page entry size:** The size of an index page entry, conceptually consisting of an <attribute value, associated OID> pair, is normally very much smaller than a data object size. This means there are on the average many more entries per index page than there are objects in a data page. This substantial difference in *density* in terms of objects

[1] In the page server system, the server keeps track of all pages cached by the clients.

associated per page plays a major role in index caching, and should be taken into account during system design and be represented in simulation models.

- **Index reads:** An index-based associative *range* query performs an index *range scan*, i.e., accesses consecutive index entries over one or more index pages, to identify the set of relevant objects. Although index range scans seem similar in nature to the data page access pattern for clustered objects, an important point to observe is that index range reads always occur at the start of processing an associative range query, and may result in either clustered or unclustered data access. Investigation of index behavior must therefore take into account both clustered and unclustered indexes.

  For example, an associative query that uses a range predicate on an object attribute with a clustered B-tree index on it will first require accessing entries (and pages) of the index sequentially over the query window, which will then retrieve a set of objects grouped closely together in some set of data pages. In contrast, a range predicate on an unclustered B-tree index will first access a consecutive set of index page entries, but they will reference objects in random data pages. Thus, the number of data objects accessed per page, i.e., the *page locality* [5], 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 index access and update costs are not considered.

- **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.

- **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 update references may follow that of data page references for object writes. Thus, index page write references are generally interspersed with data page read/write references in a transaction, even when the transaction does not access data via an index (i.e., does not have any associative queries or updates). These distinctive index page reference and update patterns have not been considered in previous studies of client-side caching.

3.3 Costs and Benefits of Index Caching

Due to the much smaller size of an index entry compared to a data object, one index page generally relates to many more data objects compared to that contained in a data page. Index pages are therefore often the most heavily used *hot spots* in a database. 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 scans and reuse of index pages, 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, and their expected costs and benefits are described in the following section.

4 Centralized and Distributed Schemes for Index Management

Keeping in mind the above issues, we define below two different schemes, one centralized and the other distributed, for access and maintenance of 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 [9]) 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 range query to the server[2]. The server scans the relevant index pages to determine which data objects fall within the specified index 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, requesting the server for the data page of an object missing from its local cache as necessary. Similarly, 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.

This 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 centralized maintenance of the index simplifies coordination of simultaneous updates to index pages by different clients. A centralized scheme may or may not work well in a client-server environment, depending 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. Note that inter-transaction caching of pages also implies their intra-transaction reuse. Which index pages get cached locally depends on the data access and update pattern of the client. Cached index pages may be read locally

[2]We consider only linear range queries in this paper.

by the client to answer index-based queries, but updating them requires coordination through the server.

Notice that the issue of controlling concurrent operations on index pages is orthogonal to the consistency control protocol adopted for data pages. In a centralized scenario, an index page is normally latched at the server (e.g., via a semaphore) only for the duration of an actual insert or delete operation on it. Locking an index page for the entire duration of a transaction is generally considered infeasible because of the cost incurred in blocking other readers and writers of the page. This would cause many deadlocks and transaction aborts, due to the large number of object entries per index page. For centralized index usage using latches, the isolation model for concurrency control is that of cursor stability [8] — there is no protection against *phantoms*, and no *repeatable read* property for queries.[3] We outline below a distributed index management scheme for a client-server environment that has a comparable isolation model.

In the distributed index management algorithm adopted in this paper, a "distributed write latch" must be obtained from the server before an index page can be updated. Acquiring such a latch involves invalidating cached copies of the index 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 client 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. Multiple readers are allowed, but index writes require exclusive ownership of a distributed latch on the index page.

The caching policy adopted for index pages is as follows. 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. For writes, even if a page is locally available, a round-trip to the server is necessary for getting a distributed write latch for the page. When the write latch is granted, a

[3]We do not consider advanced techniques such as index range locking or predicate/granular locks [8] for this study.

copy of the index page is sent over unless it is already cached locally. After the client completes the local write, it sends the updated index page pack to the server releasing the write latch, but continues caching the copy of the page. The cached copy may remain in the local buffer until it is flushed in response to an invalidation message from the server, or until it 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.

5 Simulation Model

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

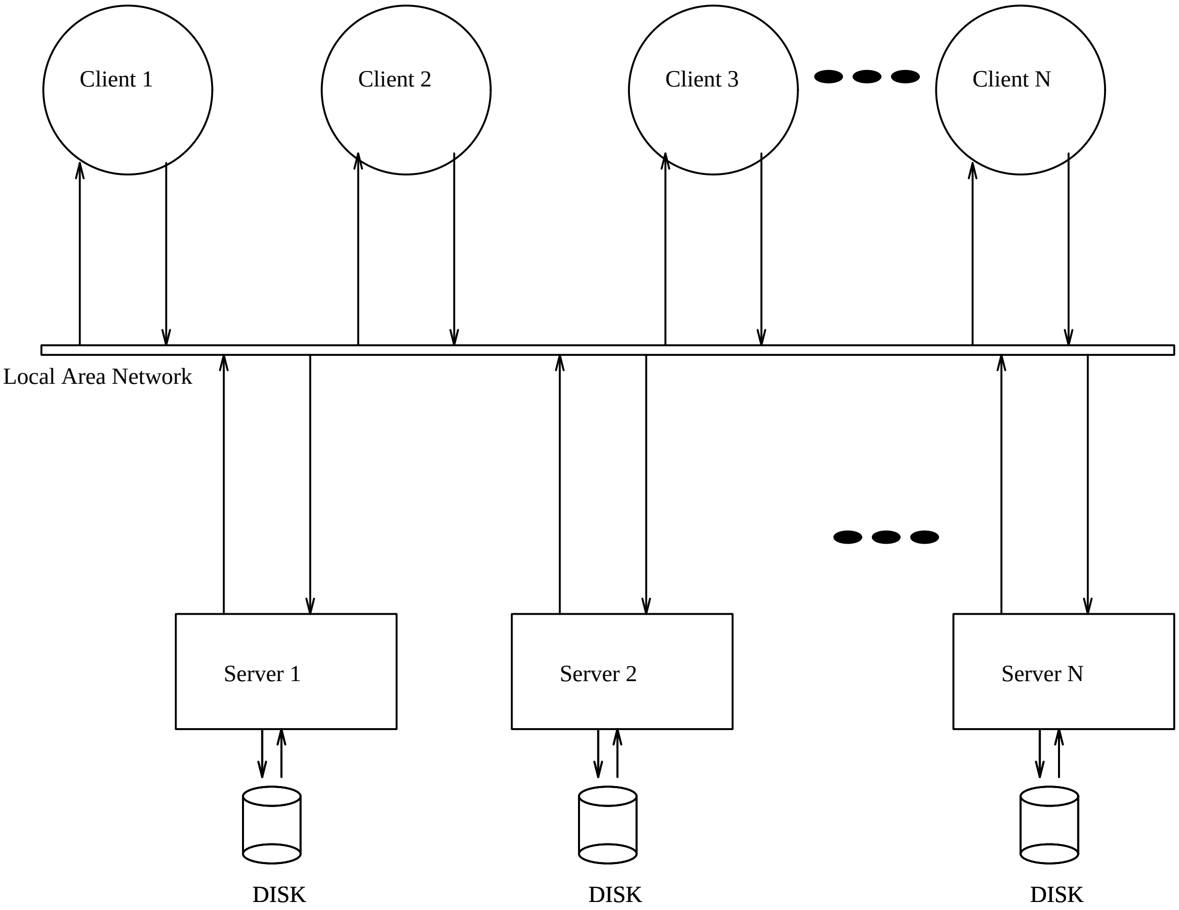


Figure 1. A typical client-server configuration communicating via a LAN.

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].

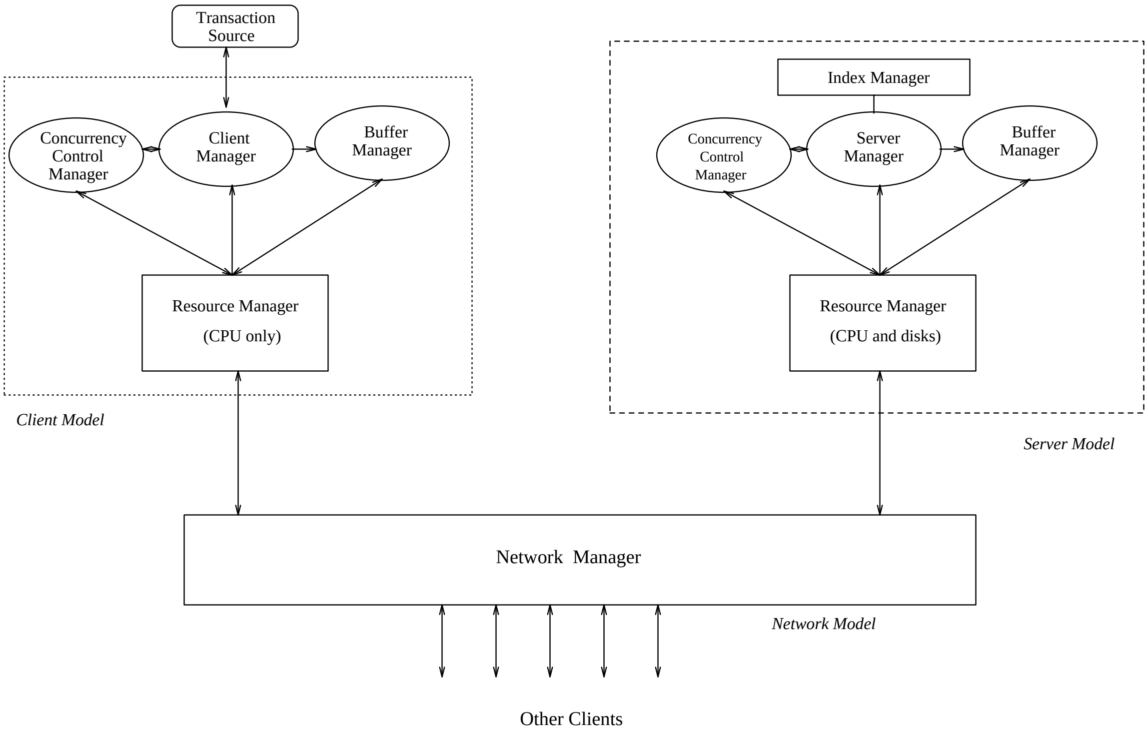


Figure 2. Simulation Model of a Page Server OODBMS

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 [5] 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 protocols, centralized and distributed, as defined earlier. Index pages are allowed to be cached in the distributed index scheme like data pages, 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

Our simulator explicitly models two indexes, one clustered and one unclustered, on the database. It is important to consider both types of indexes, since the index usage and data access patterns are different for the two. For example, 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 a unclustered index will result in random data pages being fetched in. Update probabilities may also differ for the two types of indexes — clustered indexes are generally less likely to be updated, because data objects are generally clustered in pages according to an infrequently-updated indexed attribute value, such as by the department number of employee tuples in an employee relation.

We model each associative access as an index range read, the width of the "read window" depending on another parameter — the 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 ranges are generated randomly, based the index page usage profile supplied for the particular client (workload profiles are described in detail in Section 6). The result of an index scan is a list of OIDs that are then processed by the client transaction, with object and index writes occurring according to the specified workload.

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

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. However, the client caches only load in data pages, a fraction from amongst the hot pages for the client, and the remaining from the rest of the database. No index pages are therefore cached by the clients at the start of simulation.

For each index range read, the centralized scheme makes a 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, an index update may occur may happen based on the index write probabilities per object update (a parameter in our simulation model). For the centralized case, this causes a round-trip to the server to update the appropriate indexes associated with the modified attribute value.[4] 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 actually updated (the old entry deleted and a new entry inserted) to reflect the data object write. The target page for a new entry is selected randomly from the set of all index pages of the given index. As discussed earlier, we do not consider index page splitting or

[4]Data object modifications are not sent to the server until a transaction commits or aborts.

merging; we therefore assume that an index page always has enough space to hold a newly inserted entry. All index writes are undone upon transaction abort (resulting from deadlocks, which are detected by the simulator). Since appropriate locks are held on the data objects themselves, updating the index before a transaction commits does not cause error — if a transaction reads an uncommitted index entry, it will subsequently block upon a read or request for the associated data object until the updating transaction commits or aborts.

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 [12], and uses for the basic page server the same code as the simulator developed in [3]. Thus, we have the exact behavior of data page caching as reported in [3], with index page reads, writes and local index caching being our newly added functionality.

An index page is assumed to contain 250 object entries at the outset, compared to 20 objects per data page. Given our page size of 4K bytes, this implies an index entry size of 16 bytes. For our case of 25,000 database objects, the clustered and unclustered index occupy a total of 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 a latch (of cost IxLatchCost) on the updated page(s). The actual update is assumed to have a cost of IxPageUpdateCost. As mentioned before, we do not consider index page splitting 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.

Table I: General System and Cost parameters

Parameter	Meaning	Value
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
NumClients	Number of client workstations	10
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 pages	100
NumUixPages	Number of unclustered index 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

5.4 Workload Model

Transactions in our system model are of two types: non-associative and associative. The former type of transaction does not access data using an index, and is represented as a string of data object references, exactly as 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 index interval on either the clustered or the unclustered index. Processing of such a transaction commences by examining the necessary index page(s), and by making a range scan of over these pages to generate the string 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.

The workload parameters are summarized in Table 3. The first set of four parameters are kept fixed while the second set of parameters are varied for our experiments. The third set of parameters describe the distribution of data and index pages among clients, and are also kept invariant for each client.

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.

The probability that a transaction accesses data via an index, i.e., the probability that a transaction is associative, 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 on data object updates. Given an associative transaction, CixAccProb denotes the probability that data is accessed via the clustered index. The corresponding parameter UixAccessProb for the unclustered index is simply $(1 - CixAccProb)$.

An additional parameter for each 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,

Table III: Workload Profiles

Parameter	**Meaning**	**Value**
ThinkTime	Mean think time between transactions	0
PageLocality	No. of objects accessed per page by a non-associative transaction	1-7 (min-max)
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 accesses data via an index	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 non-associative 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
CixWrtProb	Probability of clustered index update per object write	varies
UixWrtProb	Probability of unclustered index update per object write	varies
HotDataPgs	Range of hot data pages for client i	h *to* $h+124$, $h = 125*(i-1)+1$
ColdDataPgs	Range of cold data pages for client i	rest of DB
HotCixPgs	Range of hot clustered index pagesfor client i	c *to* $c+9$, $c = 10*(i-1)+1$
ColdCixPgs	Range of cold clustered index pages for client i	rest of cix pgs
HotUixPgs	Range of hot unclustered index pages for client i	all uix pgs
ColdUixPgs	Range of cold unclustered index pages for client i	all uix pgs
AccHotDataProb	Probability of accessing a hot data page	0.8
AccHotCixProb	Probability of accessing a hot clustered index page	0.8
AccHotUixProb	Probability of accessing a hot unclustered index page	uniform

CixReadSize, and UixReadSize. The first two are applicable only for non-associative transactions, while the other two are relevant only for associative transactions. NumPages denotes the mean number of pages accessed by a non-associative transaction. The size of a non-associative 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. That is, a non-associative 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, 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.

A read-write transaction chooses to update a data object with probability ObjWrtProb.[5] A data object write may result in updates to one or more 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. Updating an index entry requires deleting the old entry, i.e., the <attribute_value_before_update, OID> pair, and inserting a new entry <attribute_value_after_update, OID>. 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.

The 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 clients. Each client has its own set of 125 hot data pages, access to which occurs with a probability of 80%. Each index consists of 100 pages. The

[5]We do not consider object deletes and inserts in this study, although they could be incorporated in the simulation model.

hot bounds for the clustered index pages for each client matches its hot data page bounds — thus, each set of 10 clustered index pages corresponding to the hot data page range for the client is considered the hot clustered index page range for the client. Probability that an index range read hits the hot index page range is 80%. On the other hand, access to the unclustered index pages is assumed to be uniformly random for the 100 uix pages.

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 throughput in transactions per second (TPS) as our main measure of system performance. Additionally, our simulator keeps track of several other quantities including 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 three parameters that can be varied: AssocProb, CixReadProb, and transaction size (CixReadSize/UixReadSize for an associative transaction, and NumPages for a non-associative one).

Figure 3(a): Read-Only Clustered with Varying Transaction Size

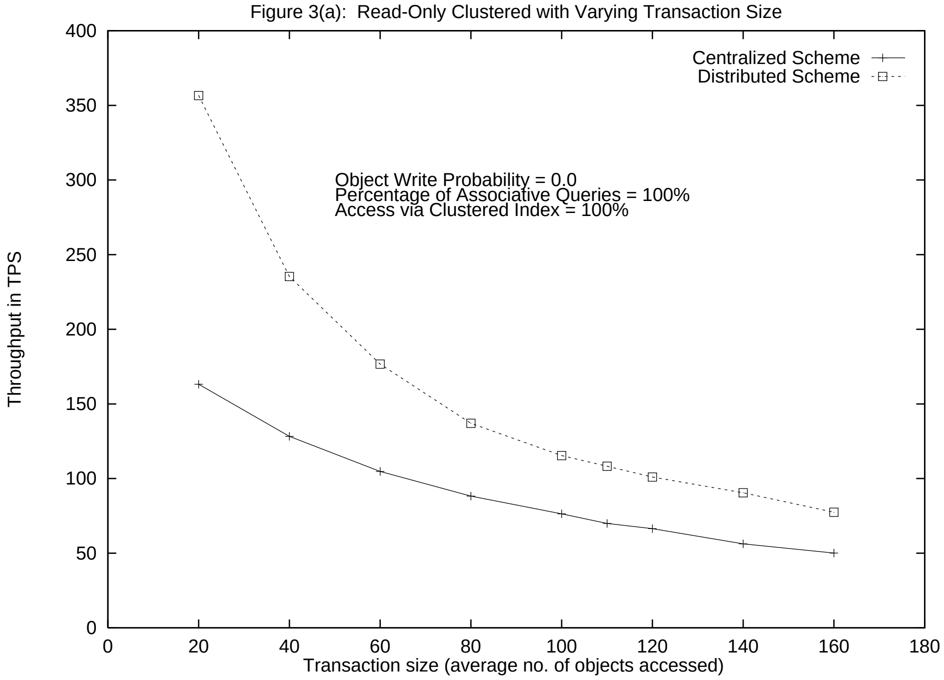


Figure 3(b): Read-Only Clustered with Varying Transaction Size

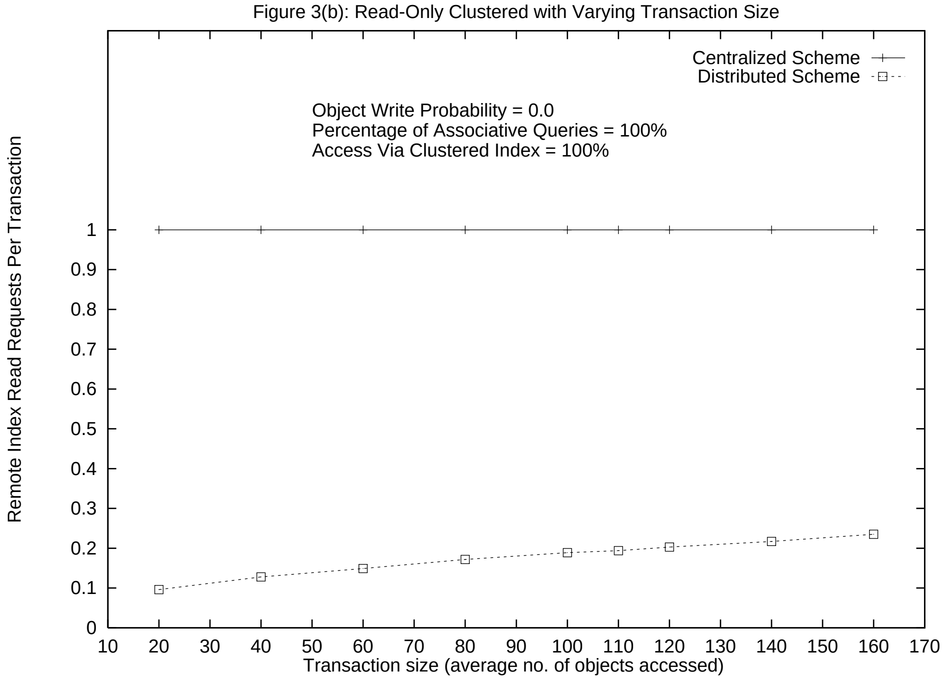


Figure 3(a) shows the effect of varying transaction size on the throughput of the system for the case where there are no object writes, all transactions are associative, and all range reads are done on the clustered index. It can be seen from the graph that the distributed scheme does very much better than the centralized for a wide range of range read sizes. This reason for this behavior is explained by Figure 3(b), which plots the number of index range 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 causes a remote index read only if there is a cache miss for the index page, and therefore the average number of remote index reads is less than 0.25 over a wide range of transaction sizes. This number is 0.1 for a transaction size of 20, and rises slowly to about .22 for a transaction size of 160.

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 range read interval overlaps two or more index pages. As the average index read window 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 larger number of data page caching per transaction, causing some index pages to be aged out of the cache and decreasing the cache hit ratio for index pages.

Figure 4(a): Read-Only Unclustered with Varying Transaction Size

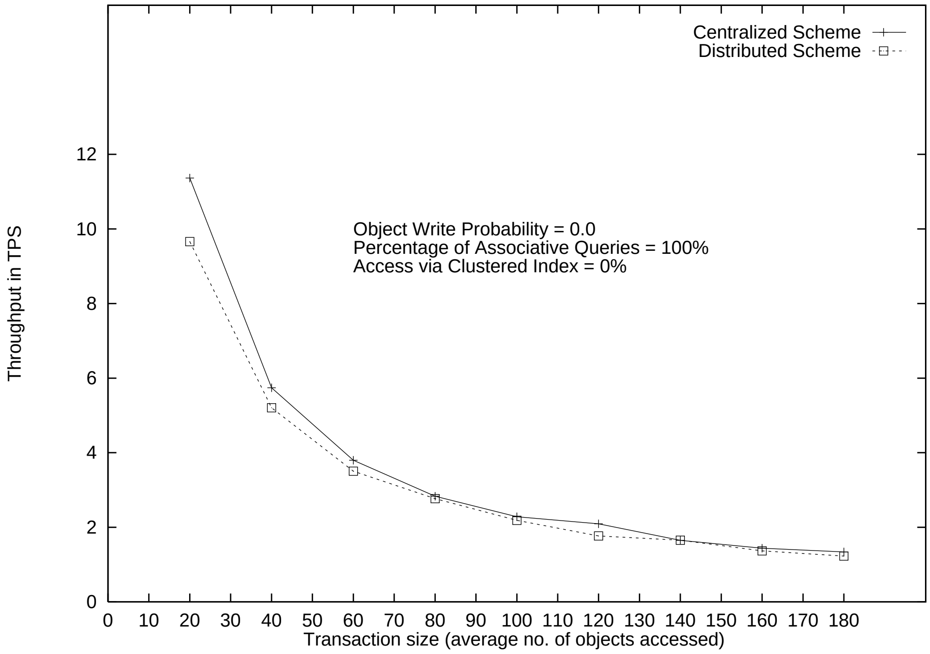
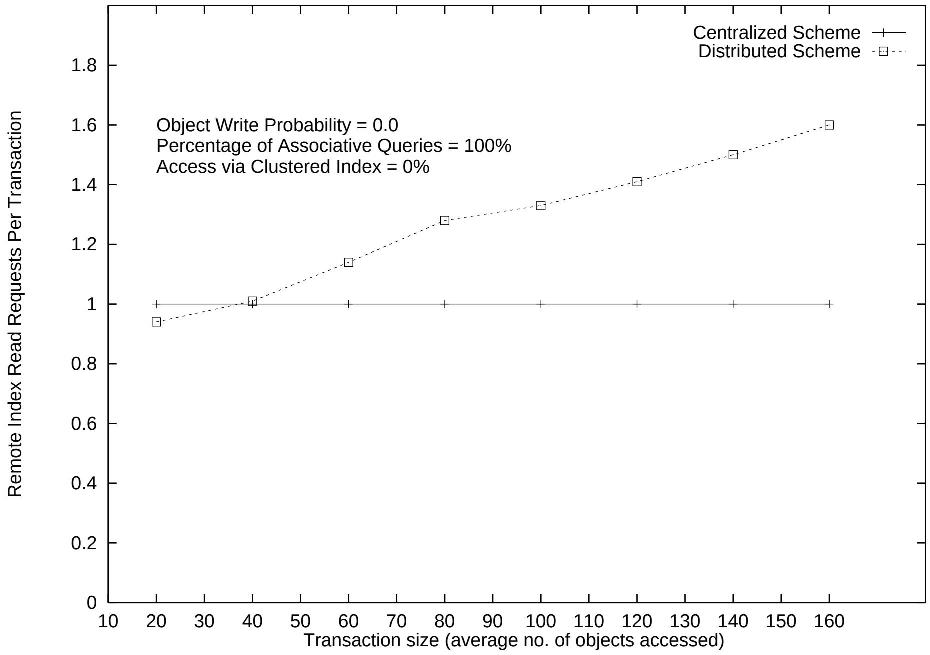
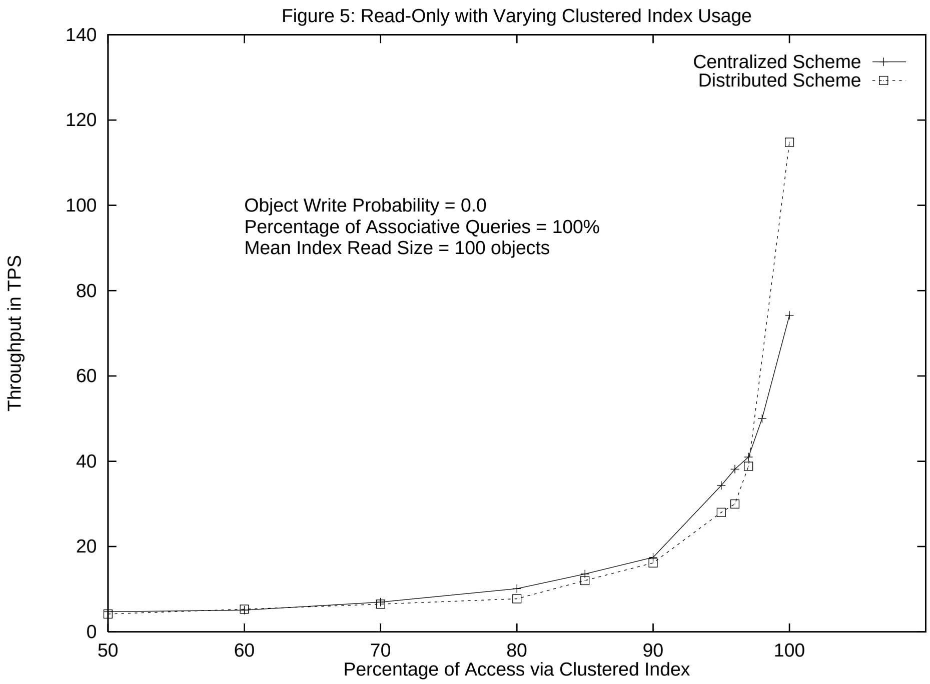


Figure 4(b): Read-Only Unclustered with Varying Transaction Size



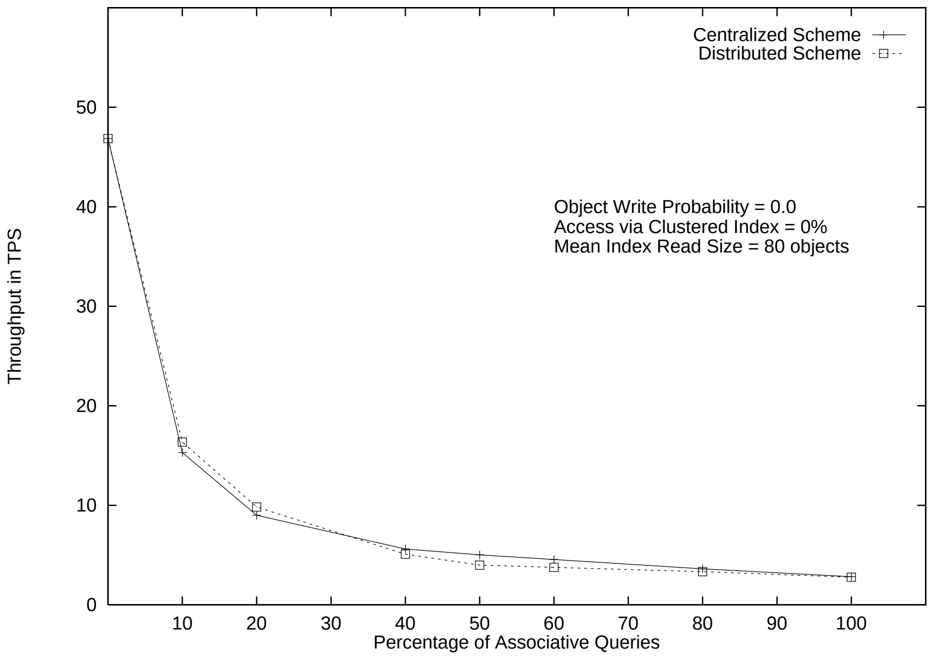
Now we consider access via the unclustered index — Figures 4(a) and 4(b) represent the same scenario as in Figure 3, but for unclustered index reads. In this case, the distributed scheme actually performs worse than the centralized, converging to about the same performance for large transaction sizes. The reason for this behavior is the our workload profile, which assumes uniformly random access to all 100 unclustered index pages by all clients. This increases the chances of a cache miss for an index read on an unclustered index page.

Figure 4(b) is the counterpart for Figure 3(b) in the case of unclustered index reads. Notice that the number of remote index page reads per transaction for the distributed scheme is much larger than in the clustered case, actually crossing the 1 constant read for the centralized scheme. One reason for the larger number of index reads is that our workload assumes all 100 unclustered index pages are accessed uniformly, making a cache hit less likely for an unclustered index page compared to the hit rate for the 10 "hot" clustered index pages for each client. The other reason is that as in the clustered index case of Figure 3(a) and 3(b), larger transaction sizes cause more data page caching activity, and age out index pages from the cache. Observe that with small transaction sizes the number of index reads is 0.94, which is less than the centralized value. Smaller transactions cause fewer caching of data pages per transaction, thereby increasing the cache hit ratio for cached index pages at the client site.



Next, consider the read-only case with varying CixReadProb, i.e., varying usage of the clustered index for range reads. Figure 5 deals with this scenario for a transaction size of 100. Notice that the distributed scheme lags slightly behind the centralized one, until about 90% clustered index access is reached. After that point the throughput of the system is better for the distributed scheme. Although this paper presents only the graph for 100-object sized transactions, we have explored the behavior of the two schemes for several different transaction sizes. For transaction lengths in the range 70 through 110, similar cross-over points were observed, with the cross-overs all occurring above the 85% value for CixAccProb.

Figure 6(a): Read-Only Unclustered with Varying Associative Queries



The last read-only scenario, i.e., varying the percentage AssocProb of associative queries, is explored in graphs 6(a), 6(b), and 6(c). In figure 6(a), all accesses are via the unclustered index and mean transaction size is set to 80 objects, while the ratio of associative queries is varied. The distributed scheme starts off slightly better than the centralized for low values of AssocProb, but falls off slightly at around 30% AssocProb. The average number of remote index reads for the setting of Figure 6(a) is presented in Figure 6(b). The number of remote index reads for the distributed scheme increases as AssocProb is increased. In contrast, the average remote index read requests per transaction for the centralized scheme increases almost linearly with the number of associative queries.

Figure 6(b): Read-Only Unclustered with Varying Associative Queries

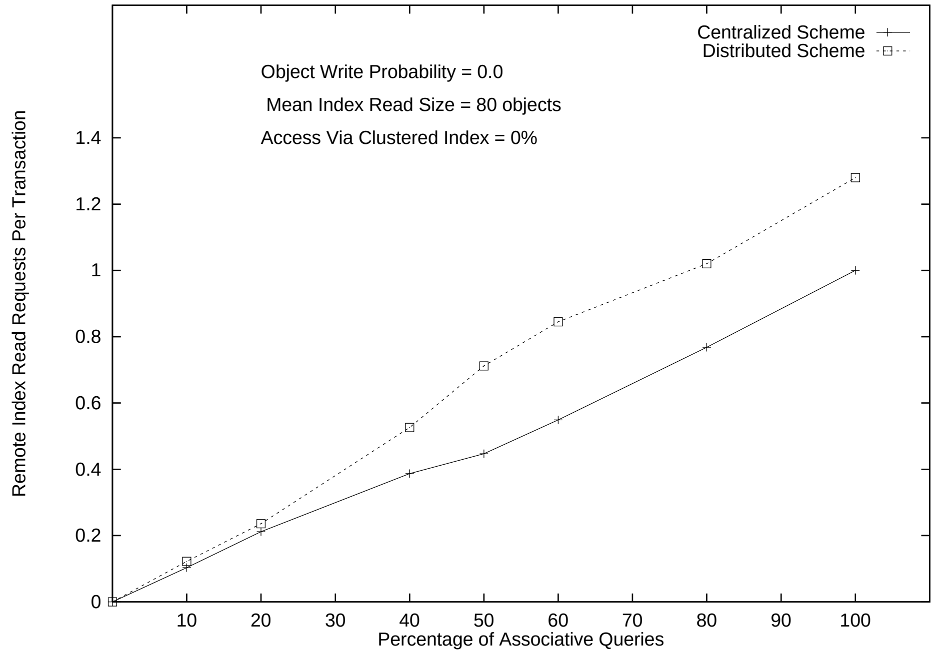


Figure 6(c): Read-Only 50% Unclustered with Varying Associative Queries

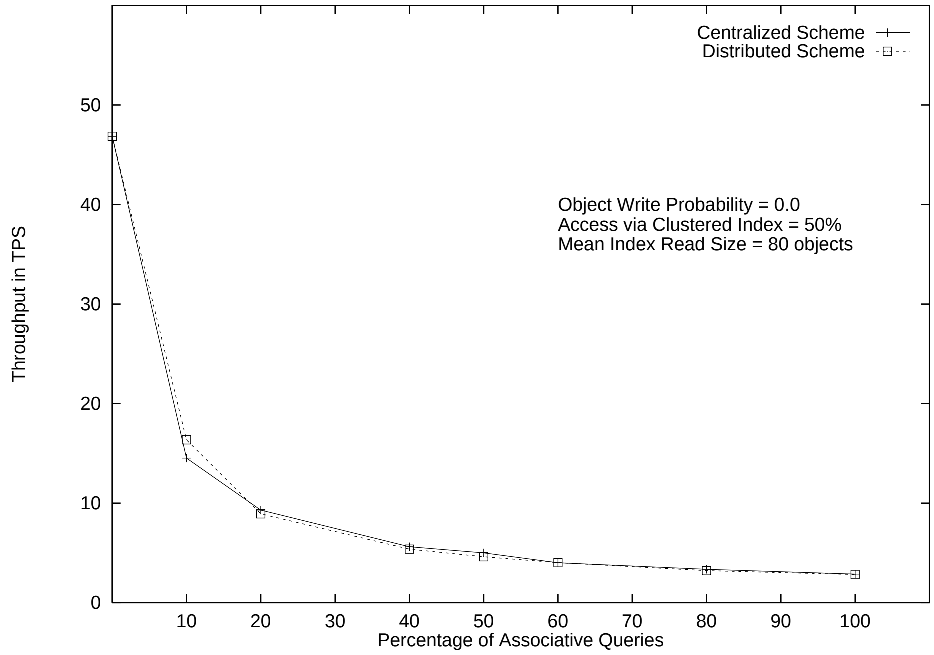


Figure 6(c) has the same parameter settings as Figure 6(a), except that usage of unclustered index is reduced to 50%. In this mixed clustered and unclustered read-only access case also, the distributed scheme quickly loses its initial advantage over centralized index reads.

6.2 Read-Write Cases

We have performed several sets of experiments varying the four major write parameters in our model, namely, ReadOnlyProb, ObjWrtProb, CixWrtProb, and UixWrtProb. All the 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 the distributed indexing scheme tends to perform significantly worse than the centralized for quite low object and index write probabilities, even with low index range reads. This behavior is not surprising given the cost associated with acquiring a distributed latch. However, we had expected in most cases to find cross-over points of the two schemes. Due to the quick deterioration in performance of the distributed scheme even for low write probabilities, cross-over points were not obtainable for many scenarios.

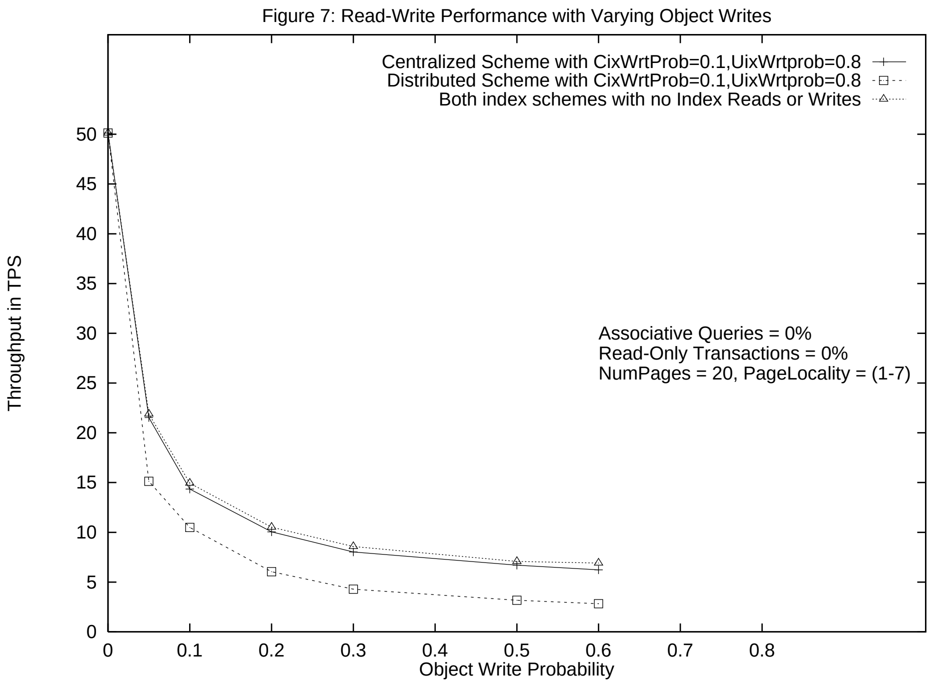


Figure 7 above shows the system performance in terms of throughput for a load of exclusively non-associative queries and mean transaction size of 80 objects. Notice that the centralized scheme for the case of 10% clustered index update probability and 80% unclustered index update probability closely 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.

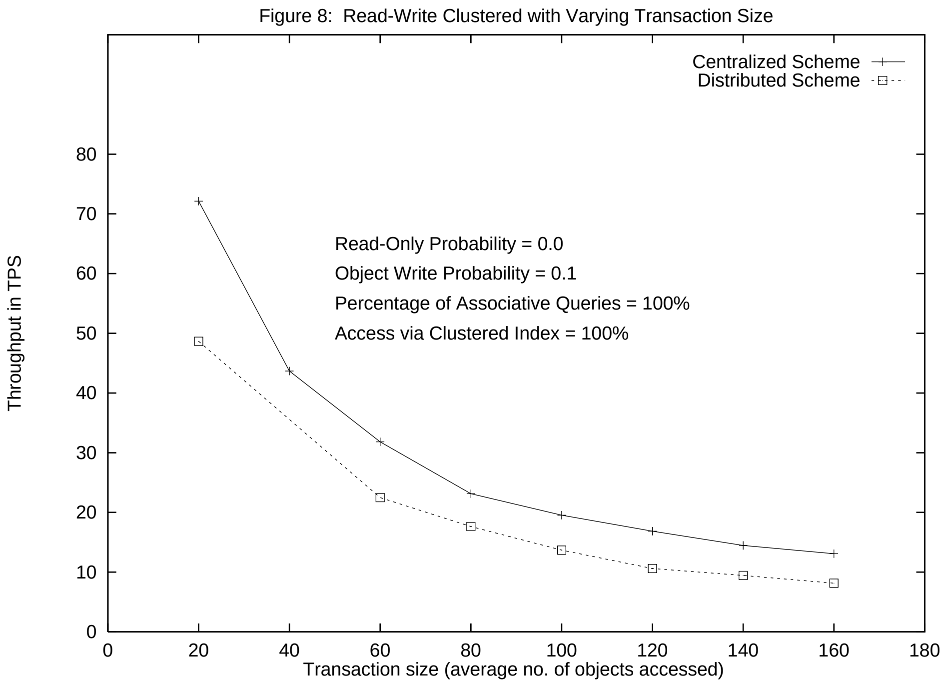


Figure 8 is the read-write counterpart for the read-only scenario of Figure 3(a). Observe that the object write probability is only 10%, but the distributed scheme performance trails the centralized one for all the transaction size settings of 20 through 160.

6.3 Mixed Load Behavior

Our model allow for associative and non-associative transactions, clustered and non-clustered access, as well as separate object and index write probabilities. We have explored mixed load scenarios where each of the above parameters has some intermediate value in its allowable range. The distributed scheme performs worse than the centralized scheme whenever there are any significant number of object writes. Some of our results are reported below.

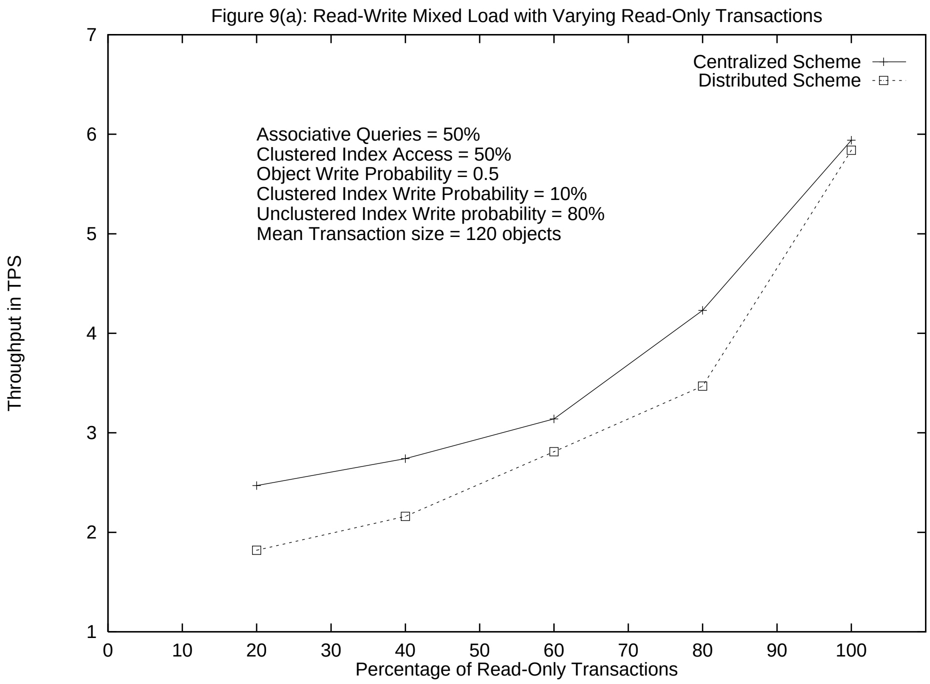
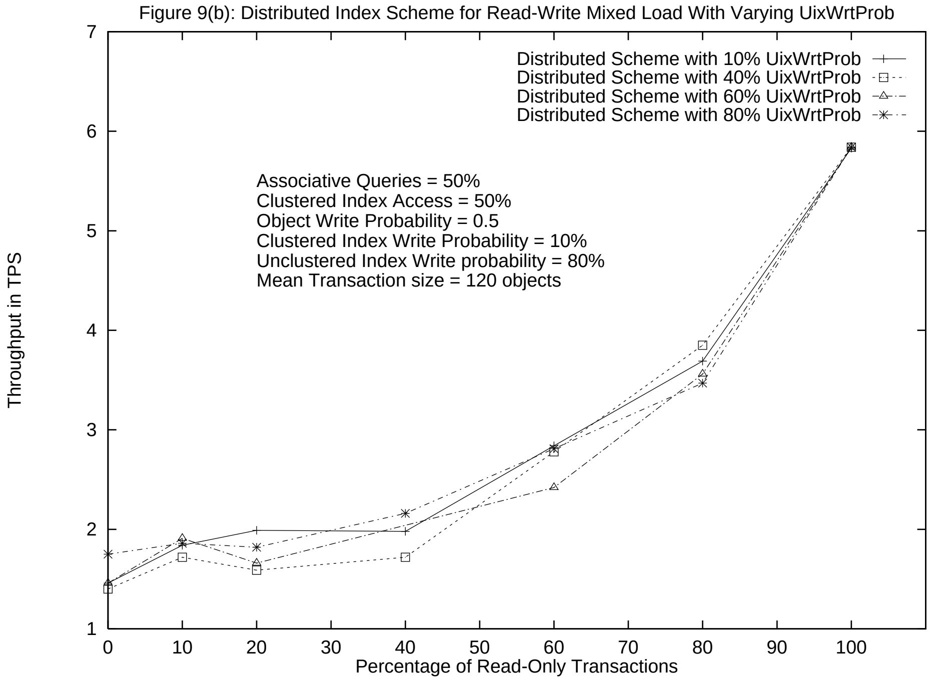


Figure 9(a) shows a mixed load with the percentage of read-only transactions being varied. The distributed scheme performance trails that of the centralized, although the relative difference in performance is not very large. Figure 9(b) reports the performance of the distributed index scheme under varying probabilities for unclustered index writes. Notice that under the given load profile, the system throughput does not vary substantially even though UixWrtProb ranges from 0.1 to 0.8.



7 Conclusion

Previous studies in client-side caching for page server OODBs have not specifically considered index access and update costs, or index page caching in particular. We have incorporated indexes and associative (range) queries in the simulation model, and evaluated through detailed simulation experiments the effects of index page caching at client sites. We compared two index management schemes, one centralized and the other distributed, that employ different approaches to index reads and writes. We have also investigated the effects of clustered versus unclustered index reads and updates for both the index schemes.

Reviewing our results, we find that distributed indexes work better for mostly read-only scenarios and when an overwhelming portion of associative access (90% or more) is through the clustered index. In other read-only cases, centralized indexes are competitive with distributed indexes. Whenever there are data and index updates and a small percentage of read-only transactions, the centralized index case performs better than the distributed index. However, depending on the workload, the difference in system throughput for the distributed scheme is not always prohibitive in nature, so that a distributed index caching scheme may indeed prove to be beneficial in certain read-intensive scenarios.

In conclusion, indexes behave very different than data in a page server OODBMS. While distributed query processing appears profitable for data pages in a page server OODBMS, centralized (i.e., server-based) index processing seems superior in many cases. 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.

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 Trade-offs in Client-Server DBMS Architecture," *ACM SIGMOD Int. Conf. on Management of Data*, Denver, CO, May 1991.

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

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

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

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

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

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

[9] 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.

[10] A.M. Keller and J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures," *Third Intl. Conference on Parallel and Distributed Information Systems*, Austin, TX, Sept. 1994.

[11] 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, p. 510-22.

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

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

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