Relaxed Index Consistency for a Client-Server Database*

Vibby Gottemukkala
IBM T.J.Watson Research Center
Yorktown Heights, NY 10598

Edward Omiecinski   Umakishore Ramachandran
College of Computing
Georgia Institute of Technology, Atlanta, GA 30332

Abstract

Client-Server systems cache data in client buffers to deliver good performance. Several efficient protocols have been proposed to maintain the coherence of the cached data. However, none of the protocols distinguish between index pages and data pages. We propose a new coherence protocol, called Relaxed Index Consistency, *that exploits the inherent differences in the coherence and concurrency-control (C&CC) requirements for index and data pages. The key idea is to incur a small increase in computation time at the clients to gain a significant reduction in the number of messages exchanged between the clients and the servers. The protocol uses the concurrency control on data pages to maintain coherence of index pages. A performance-conscious implementation of the protocol that makes judicious use of version numbers is proposed. We show, through both qualitative and quantitative analysis, the performance benefits of making the distinction between index pages and data pages for the purposes of C&CC. Our simulation studies show that the Relaxed Index Consistency protocol improves system throughput by as much as 15% to 88%, based on the workload.*

1 Introduction

The Client-Server architecture has become the dominant database system paradigm because of the proliferation of inexpensive, high-performance desktops that can perform complex database operations on data that is stored on server(s) for the purposes of sharing and persistence. Client-caching of data and concurrency-control information is a commonly used technique to reduce the amount of interaction between nodes. However, caching introduces the problem of maintaining coherence of multiple copies of data to ensure correctness. Several techniques, both pessimistic and optimistic, have been proposed to efficiently maintain coherence of data and ensure transaction serializability. The major drawback with these techniques is that they treat all the data accessed by transactions in the same uniform manner and thus fail to exploit the performance advantages that can be derived by recognizing the semantics of access to different types of data within a database.

Assuming that data is accessed as pages, the major distinction we make between the different pages in a database is whether those pages are *data pages* or *index pages*. Transactions access both data and index pages, but the consistency requirements for data and index pages differ. In general, index pages are accessed more often, and hence cached (replicated) at more clients, than their corresponding data pages. Furthermore, assuming tree-based indexes (*e.g.* B+-tree, Blink-tree), index pages at the root and intermediate-node level are accessed more often (and hence, replicated more) than index pages at the leaf level. Typically, invalidation or update-based coherence maintenance protocols are expensive when the degree of replication of pages is high and pages are accessed often, as is the case for index pages. It is our objective, therefore, to devise an efficient coherence protocol for index pages that takes advantage of the semantics of their access.

In this paper, we propose and evaluate a new protocol, called *Relaxed Index Consistency* (RIC), that distinguishes between data and index pages. In RIC, index operations may traverse an out-of-date index but always return consistent data. Furthermore, the coherence maintenance for index pages is integrated with the C&CC operations on data pages to reduce the internode interaction. Thus, explicit concurrency control for index pages is eliminated for the general case. The key idea is to trade-off a little bit of additional processing at the clients for a significant reduction in client-server communication. A quantitative comparison between RIC and the Callback Locking protocol shows that RIC provides transaction throughput improvements between 15–88%, based on the workload and percentage of writes. Furthermore, RIC reduces the amount of coherence activity between nodes by as much as 65–75%, and the number of messages by 25–35%.

The rest of this paper is organized as follows: In Section 2 we present the details of our approach and compare it with those in the literature. In Section 3 we discuss implementations of our protocol and of index operations using our protocol. In Section 4 we qualitatively and quantitatively analyze the performance of RIC in comparison with Callback Locking. Finally, in Section 5 we summarize our contributions and draw our conclusions.

2 Design Overview

The overall objectives of our design are to improve the performance and scalability of the database system. More specifically, we would like to reduce the number of messages that are exchanged among the clients and the server, the reasons for which are two fold. First, although current network technology trends indicate that the bandwidth of networks keeps increasing, the latency (due to software overhead) for individual messages continues to be high. Second, in Client-Server architectures, the clients have to necessarily interact through

*This work was supported in part by an NSF PYI Award MIP-9058430, and matching grants from DEC, SYSTRAN, and IBM, and was done when V. Gottemukkala was at Georgia Tech. Contact:*vibby@watson.ibm.com*

the server, thus making the server a bottleneck to scalability. Therefore, our objective is to reduce the number of global concurrency control operations, and the amount of coherence activity between nodes. We expect that these guidelines would help us reduce the number of messages in the system, and also allow the server to handle more clients simultaneously, thus improving the performance and scalability of the system. In the following, we outline our approach to satisfying these guidelines.

2.1 Consistency requirements

C&CC protocols can be broadly classified into optimistic and pessimistic protocols. As pointed out in [15], there are several practical problems in implementing optimistic protocols, such as space management, access path maintenance. On the other hand, using a straightforward pessimistic C&CC protocol for both index and data pages would make coherence maintenance for index pages expensive because of their high frequency of access and degree of replication. Thus, within the framework of pessimistic C&CC protocols, we would like to explore ways of making index coherence less expensive in order to meet the design goals stated earlier. In the following we discuss aspects of index operations and concurrency control that allow us to achieve this objective.

All index operations (*e.g.* scan, insert, delete) can be divided into two phases. In the first phase, the operation traverses the search structure to locate the position where the operation has to be performed. We call this phase the "positioning phase." In the second phase the actual operation is performed and we call it the "operation phase." In the operation phase logical locks are acquired on the index items of interest in order to satisfy the consistency requirements of the transaction executing the index operation. However, there is no such restriction on the positioning phase. The only guarantee that is needed is that the operation phase operate only on the *relevant* index items. Thus, there is scope to reduce or even eliminate C&CC activity for index pages during the positioning phase so long as their consistency is validated during the operation phase.

Furthermore, for the logical locking of index items we propose to use *data-only locking*. Data-only locking implies that when an index item is locked, the lock is actually acquired on the referred-to data item (page or record lock based on lock granularity). Data-only locking significantly reduces the actual number of locks that have to be acquired by a transaction [14] which directly contributes to a reduction in the number of messages in the system.

Our objective is to use data-only locking for concurrency control and at the same time take advantage of the potential to relax the coherence maintenance for index pages. For this purpose we propose a new C&CC protocol, called *Relaxed Index Consistency* (RIC), that allows transactions to access indexes without any coherence control but performs coherence maintenance on data and indexes when logical locks on data items are acquired. In the following we present the RIC protocol and later discuss how index operations can use the protocol to satisfy our design goals.

```
............ INDEX PAGES ............
  if (page_type == INDEX)
      if (request_mode == SHARED)
          goto grant_request
      else /* (request_mode == EXCLUSIVE) */
          if page currently locked
              issue callback to locking client
              remove responding client from
              copy_list
          record exclusive lock
          goto grant_request

............ DATA PAGES ............
  else /* (page_type == DATA) */
      standard CALLBACK LOCKING protocol +
      goto grant_request

grant_request:
  if (server copy newer than requesting client's)
      include page in page_granted message
  if (page_type == DATA)
      include coherence information in
          page_granted message
  send page_granted message
```

Figure 1: Server actions under the RIC protocol.

2.2 The RIC Protocol

In our client-server model client transactions interact with a local client cache manager (CCM) to access pages or acquire locks on those pages. Both actions result in the CCM granting local locks on the pages to the transactions. In granting the local locks, the CCM may have to acquire global locks (according to the C&CC protocol) from the server. We assume that the global locking is integrated with page transfer. We also assume that the CCM caches pages (and the global locks that it may have acquired on them) beyond transaction boundaries.

When a transaction requests a page the CCM checks to see if the requested page is available in the cache and if the mode in which it is available is compatible with the requested mode. If both conditions are met, the CCM grants the requesting transaction a local lock on the page. Otherwise, the CCM generates a cache miss, that is serviced by sending a request to the server for the page in the accessed mode.

The actions taken by the server in response to a page request differ for index and data pages and are outlined in Figure 1 (we assume that the CCMs and the server can distinguish between index and data pages). We can see from Figure 1 that when the CCM requests an index page in shared mode the server does not acquire any locks on its behalf. Therefore, when an index page is accessed in exclusive mode the only potential coherence related communication (callbacks) is with another client that may have the page cached in exclusive mode. In fact, the server does not even keep track of clients that cache index pages in shared mode. As for data pages, the server uses an enhanced Callback Locking (CBL) protocol [4, 20] to grant a client's page request. The CBL protocol is enhanced for RIC so that the server, in addition to granting the requested data page, sends coherence information for the indexes relevant to the requested data page (see the `grant_request` step in Figure 1). In the case where the CCM gets index coherence information from the server, it uses the information to determine if the transaction accessed any inconsistent index pages and indicate it to the transaction so that it may take the appropriate action. The exact nature of the coherence information and its usage are discussed in later sections.

Callbacks: We have seen that the server may issue callbacks to clients for both index and data pages. When a CCM receives a page callback it checks if the page is locally locked and sends a successful response to the server only when the page is locally (eventually) unlocked. In the case of data pages, the CCM marks the called back page invalid. However, for index pages the CCM continues to cache a copy of the index page in shared mode even though a callback on the page implies that the page is going to be modified. The reason for this caching is to allow index operations that access index items that are not affected by the modification to proceed without having to re-acquire a copy of the page from the server.

2.3 Using the RIC Protocol

Index operations can use the RIC protocol to exploit the two-phase nature of index operations outlined earlier. In the positioning phase the index operation accesses pages in shared mode. Therefore, from the RIC protocol description we know that these pages are either available in the local cache or obtained from the server, but no coherence actions are taken. Thus, the positioning phase could potentially access an inconsistent index. However, in the operation phase logical locks are obtained on the items that are of interest to the index operation. When a logical lock is granted, either locally or by contacting the server, RIC has to guarantee that the locked index item belongs to a consistent index. In order for RIC to provide this consistency guarantee, all index operations have to satisfy the following two constraints:

1. All index operations must acquire at least one logical lock on an index item.
2. Between the modification of an index page by one client and a subsequent access of that page by another client, the latter client must acquire at least one logical lock that causes communication with the server.

In the following we discuss, through an example, how these constraints are met by index operations and how RIC ensures their consistent execution when the constraints are met. We assume, for the sake of this discussion, that the transactions execute at Isolation Level 4 ('Serializable') as specified in the SQL-92 standard [12] which requires index operations to support repeatable reads (RR) and avoid phantoms [7, 13]. Index operations typically meet this requirement by using key-range locking which involves, besides locking all the relevant index items, locking an item adjacent (previous-key or next-key) to the relevant index items. We assume that previous-key locking is used. Furthermore, we assume that a B+-Tree index is used.

We have to note here that when an index is modified by a transaction on one client, the transactions on the other clients could see one of two forms of inconsistencies in the index. One inconsistency is logical inconsistency, *i.e.,* either missing or additional index items that should or should not be there in a consistent index. The other form of inconsistency is structural inconsistency where the index structure is changed because of index page overflow or underflow.

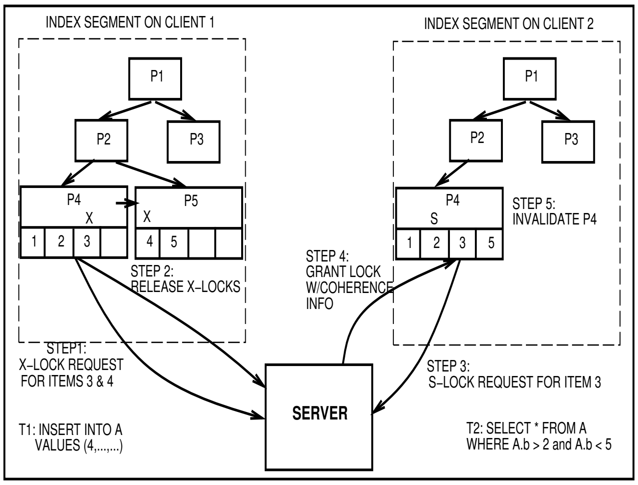

Figure 2: Example of index operations under RIC.

We consider an example where a modification on one client causes structural inconsistency (and, by implication, logical inconsistency) in another client's view of the index and examine how, using the RIC protocol, this inconsistency is detected and recovered from. Let us assume that both $Client_1$ and $Client_2$ initially cache identical images of the index, with items 1, 2, 3, and 5 in index page P4 (cached in shared mode), as shown by $Client_2$'s state in the right side of Figure 2. Consider that $Client_1$ executes a transaction (T1 in Figure 2) that inserts a tuple resulting in an index operation trying to insert item 4 on page P4. Let us assume that the insertion would cause an overflow in page P4 and results in the contents being split between P4 and a new page P5 (see Figure 2). In order to perform this insert, the index operation obtains the relevant locks, namely, X locks on P2, P4, P5, X lock on item 3 (lock on adjacent item),[1] and X lock on item 4. However, $Client_2$ which caches P2, P3, and P4 (in shared mode) is not notified. Now, consider that $Client_2$ executes a transaction that accesses P4 (T2 in the Figure 2). According to the state of P4 on $Client_2$ there is only one index item (item 3) that satisfies the index operation of T2, although we know that is not the case. T2 acquires a shared lock on item 2 as per the previous-key locking requirement, thus satisfying condition 1 of the RIC protocol of locking at least one item in the index. Then, the index operation acquires a shared lock on item 3 which results in communication with the server because $Client_1$ had previously held the lock in exclusive mode. Thus, the lock request for item 3 satisfies condition 2 of the RIC protocol. When the server grants the lock on item 3 to $Client_2$, it also includes information regarding the state of the index pages. $Client_2$ upon receiving

[1] Note that if $Client_2$ had the logical lock on item 3 cached, it would be called back before being granted to $Client_1$. Furthermore, since we are using data-only locking, the locks on the index pages are different from the locks on the index items (which are actually locks on the referred-to data items).

this information checks the validity of the index and the index operation is notified. The index operation then finds that page P4 of the index is invalidated (Step 5 in Figure 2) and repositions and restarts the scan. When the scan is restarted, first a valid copy of P4 is obtained and item 3 is returned. Furthermore, when the index operation continues and tries to access the next item that satisfies the predicate, it obtains a copy of page P5 (a pointer to which is stored on P4) and accesses item 4, thus completing the operation correctly and successfully. Detection and recovery of inconsistencies in other index operation cases is discussed in [6].

From the discussion we can see that the RIC protocol eliminates multi-point communication for index page coherence at the risk of extra processing for scan repositioning. In summary the RIC protocol eliminates physical locking for read/shared access to an index page but serializes modifications through a physical lock, thus allowing a single writer and multiple readers to exist concurrently. The RIC protocol ensures consistency of an index when an index item is locked using data-only locking thus combining coherence actions on index pages with C&CC of data pages.

2.4 Related Work

There is an extremely large body of work in the literature regarding coherence and consistency maintenance of data in various data sharing environments [3, 22, 20, 1, 17]. However, this entire body of work treats all the data uniformly, *i.e.,* the same protocol is used for both data and non-data pages, which leads to unnecessary inter-node interactions in the case of non-data pages. There is also a large body of work regarding coherence and consistency maintenance of indexes in a distributed setting. In [21] the general dictionary lookup/maintenance algorithm for coherent shared memory is transformed to run on a software cache coherence scheme called multi-version memory which provides weaker semantics than coherent shared memory. The multi-version memory supports a set of read, write, and synchronization operations that allows multiple readers (reading older versions) and single writer to exist simultaneously. However, for each node in the search path the current copy has to be obtained before proceeding with the search. Furthermore, explicit locks have to be obtained on each of the leaf nodes. Both these restrictions are eliminated in our scheme where we exploit the semantics of the concurrency control operations that are used to enforce transaction consistency. In [9, 10] the authors propose index maintenance schemes in a distributed memory environment. In this environment leaf nodes of an index are not replicated but instead are owned by nodes, and operations that need to be performed on a leaf are performed on the owner node by sending it a message. Only the interior nodes of an index are replicated across the nodes. Hence, the coherence schemes only have to worry about the interior nodes and the authors present algorithms for lazy coherence maintenance. Furthermore, the algorithms present the index page access in isolation from the data page access and hence do not consider the possibility of the integration of C&CC as our scheme does.

In [16] C&CC protocols that allow multiple readers and a single writer simultaneously are presented, but these protocols do not distinguish between index pages and data pages. The coherence of a page is enforced when a logical lock on an item in that page is acquired. However, in the case of internal pages in indexes, there are no items on which logical locks are acquired. Furthermore, when data-only locking is used for logical locks on index items, several extensions have to be made to these protocols in order to efficiently maintain the coherence of the index pages. The RIC protocol picks up where these protocols left off by distinguishing between index and data pages and addressing the issue of maintaining coherence of index pages even though no locks are acquired on them.

The Callback Locking protocol used for data locking in RIC was originally proposed for file caching in AFS [8]. This protocol was enhanced and adapted for caching in transactional systems in [11, 20, 4]. In the implementation of CBL that we use for the RIC protocol, when a transaction requests an exclusive data lock, upon its release the client downgrades the lock to a shared lock in order to avoid excessive callbacks, because caching exclusive locks causes lock conflicts.

3 Implementing Relaxed Index Consistency

In this section we outline a possible implementation of relaxed index consistency, and enhancements and optimizations for an efficient implementation. In Section 2 we identified when coherence information needs to be exchanged between nodes, namely, when data lock requests result in communication with the server. We call these points of communication *synchronization points* (SPs). In this section we discuss what coherence information needs to be contained in the messages exchanged at SPs. Furthermore, we discuss how clients use this coherence information to ensure that index operations satisfy the consistency requirements of the client transactions.

We assume that the pages in the database are organized into *segments* based on their relationship to one another. Each segment has an associated type that identifies it as either a data or an index segment. We assume that the relationship between index segments and data segments is known.

Coherence information: In our implementation of RIC each page has a version number which could simply be that page's log sequence number (LSN). The set of all the version numbers of a segment's pages capture a node's view of the state of that segment. For each index segment, at each node, we maintain a data structure, the *segment coherence information structure* (SCIS), that reflects the local version numbers of all the pages in that segment. The coherence of an index segment on one node, with respect to another node, can be determined by comparing the two nodes' SCIS for that segment. The SCIS can be viewed as a vector of version numbers of all the pages in an index segment.[2]

The semantics of the RIC protocol require that the server send coherence information to a client at every synchronization point. Furthermore, since locks on index items in different indexes could translate to a lock

[2]Although not discussed here, the SCIS has to be a slightly more complicated structure than a vector of version numbers in order to be able to handle creation and deletion of index pages from a segment.

on a single data item, at a SP, the coherence information should include all the index segments relevant to the data item locked at the SP. The reasons for these requirements are discussed in detail in [6]. In light of these requirements, a straightforward implementation of RIC would be that at each SP the coherence information exchanged contains the SCIS of all the index segments related to the data on which the lock is acquired. However, this requirement on the content of the coherence information exchanged at each SP could result in unnecessarily large messages from the server and unnecessary validation processing at the clients. Next, we present an optimized implementation that alleviates this problem.

Optimizations: In Figure 3 we outline an optimized implementation for the exchange and usage of coherence information in RIC. In the following we discuss the optimizations over the straightforward implementation, that resulted in this optimized implementation.

We saw in the straightforward implementation that at a SP the server has to include the SCIS of all the relevant index segments. The reason for this inclusion is that the server has no information to determine if any of the relevant index segments changed since the last time the client was sent coherence information. In order to allow the server to make this determination, we associate with each index segment a *segment-version number*, in addition to the SCIS mentioned previously. The segment-version number is incremented whenever there is a modification in that segment. When a client requests the server for a data lock, the client identifies for the server the current index segment in use and its segment-version number when a data lock is acquired (Step 1 in Figure 3). When the lock is granted, the server sends the SCIS of the identified index (if necessary) and in addition sends only the segment-version numbers of the other relevant index segments. These actions are taken during the `grant_request` phase (in Figure 1) of the RIC protocol. Thus, when a data lock is requested the server would have to include at most one SCIS in the coherence information, if an index operation is in progress on the requesting client.

```
............ CLIENT ............
STEP 1:
  if (index operation is in progress)
      include segment_id of
      current index segment
      AND segment_version_num in
      data page request message
  send data page request message to server
STEP 2:
  receive response from server
  for each of the relevant index segments
      if (segment_version_num in message >
          local segment_version_num)
          invalidate local SCIS for that segment
          if (this index segment is
              current index segment)
              replace local SCIS with
                  SCIS in message
              index_inconsistency = TRUE
  if (index_inconsistency == TRUE)
      notify index operation of inconsistency
STEP 3:
  for (each page in the scan list)
      check validity of page using
          segment's SCIS
      if (page is invalid)
          return(page_id)
  return (SCAN_PATH_FINE)
```

Figure 3: Actions at a synchronization point.

3.1 Actions at synchronization points

In this section we discuss the actions that a client has to take when it receives the index coherence information that we have discussed about thus far. This involves several steps that includes both the client cache manager and the client transaction. We assume that the CCM can detect if an index operation is in progress. Step 2 in Figure 3 describes how the CCM uses the coherence information obtained at a synchronization point to validate the relevant index segments and notify the index operation in progress of any inconsistency in the index segment accessed by the operation. When the index operation receives the notification it has to decide if its operation thus far is affected because the index inconsistency could be in a portion of the index that is totally unrelated to the current index operation. To perform the check, the index operation hands an ordered list of index pages it has visited to the CCM. The CCM compares the list of pages accessed with the information in the newly installed SCIS to determine if any of the accesses were from an inconsistent index (Step 3 in Figure 3). The CCM returns the identity of the earliest page in the list that it found to be inconsistent. The index operation uses the returned index page identity to reposition its scan before continuing with the index operation. It has to be noted here that none of the operation phase already completed has to be repeated. This is due to the fact that before an item is retrieved, the previous-key locking used allows index operations detect any inconsistencies that might have occurred in the interval between the previous key being locked and the next locally available item. An example of this was seen in Section 2.3 when the shared lock on index item 3 allowed the operation to detect the inconsistency between item 3 and item 5. Thus, once an item is retrieved by the index operation, it is guaranteed to belong to a consistent index.

In Step 2 of Figure 3, we see that the CCM only validates the SCIS of the relevant index segments and not the individual pages in those segments. Later, when an index page in the cache is accessed by a transaction, the CCM first checks if the SCIS for that segment is valid. If the SCIS is invalid, then the client contacts the server to obtain the latest SCIS for that segment. The CCM then uses the same methodology as that in Step 3 of Figure 3 to check if the page is valid. If the page is invalid, the CCM requests the server for the page in the accessed mode (as described in Section 2.2).

Thus far we have assumed that all the transactions execute at Isolation Level 4 which ensures that index operations satisfy the two constraints for correct execution under RIC. In [6] we discuss how transactions executing at lower level of isolation can operate correctly under RIC.

CBL	RIC
MODIFICATION BY A CLIENT	
Excl. lock request: Client → Server	Excl. lock request: Client → Server
Callbacks: Server → N Clients	No callbacks
Responses: N Clients → Server	No responses
Grant message: Server → Client	Grant message: Server → Client
READ BY A DIFFERENT CLIENT	
Shared lock request: Client → Server	Data lock request (SP): Client → Server
Downgrade request: Server → Client	Downgrade request: Server → Client
Acknowledgment: Client → Server	Acknowledgment: Client → Server
Grant message: Server → Client (w/page)	Grant message: Server → Client (w/data page & SCIS)

Table 1: Message activity comparison of CBL and RIC protocols

4 Performance Analysis

In this section we evaluate the performance of the RIC protocol. We compare the RIC protocol with the Callback Locking (CBL) protocol because of the general acceptance of the performance advantages of the CBL protocol in the literature and its implementation in systems [4, 11, 2]. First, we present a qualitative comparison of the two protocols and later we present a quantitative evaluation.

4.1 Qualitative Analysis

Table 1 compares the two protocols in the case where an index page (cached at N clients in shared mode) is modified by a client, and later a different client reads the page. As we can see from Table 1, in CBL there are $2 \times (N+1)$ messages in the modification phase and 4 messages in the read phase. In comparison, the RIC protocol requires 2 messages in the modification phase and 4 messages in the read phase. This represents a saving of $2 \times N$ messages in the modification phase, all of which involve the server. The number of messages in the read phase is the same although in the RIC protocol the sizes of the messages at synchronization points may be larger because of the coherence information exchanged at these points. However, with the optimized implementation presented in Section 3 we know that at most one SCIS (whose size is directly proportional to the number of pages in an index segment) has to be included in the coherence information. This is in agreement with our design goal of trying to reduce the number of messages even at the cost of increasing the size of certain individual messages.

Other qualitative benefits of the RIC protocol include the elimination of explicit locking for scanned index pages and server notification for index page replacement, and the occurrence of fewer deadlock situations due to reduction in locking. As can be seen from the discussion, the RIC protocol trades off inter-node communication with slightly increased message sizes (index coherence information), additional local computation (index consistency checks), and state information (per node SCIS).

4.2 Quantitative analysis

4.2.1 Simulation model

In order to quantitatively study the performance characteristics of relaxed index consistency and compare it with callback locking, we developed a process-oriented simulation model based on CSIM [19]. In this section, we briefly discuss the system and workload aspects of our simulation.

Parameter	Value
System Parameters	
Client Processor	25 MIPS
Client Cache Size	512 pages
Server Processor	50 MIPS
Server Cache Size	1024 pages
Number of Disks	2
Min Disk Access Time	10 ms
Max Disk Access Time	30 ms
Network Bandwidth	80 MBits/sec
Number of Clients	15 or 30
Database Parameters	
Page size	4096 Bytes
Data page capacity	16 tuples
Index page capacity	128 entries
Data segment size	2048/4096 pages
Index segment size	256/512 pages

Table 2: System configuration parameters

The client-server system we model has a single server with attached stable storage and a number of clients. Client nodes have a transaction manager and a cache manager competing for the client CPU. The transaction manager generates and executes transactions that involves executing a stream of data and lock requests. The cache manager is responsible for satisfying the data and lock requests of both the local transaction manager and requests from the server. Several server processes (that simulate a multi-threaded server) handle requests from clients for data and locks and compete for the server CPU and the disks. Furthermore, both the client cache manager and the server processes compete for the network resource.

A parametric specification of our system model is given in Table 2. All the resources in our simulation model use simple FIFO scheduling. Furthermore, non-preemptive scheduling is used to allocate CPUs to processes. Processes relinquish CPU control when performing either network or disk I/O. The buffers on both the clients and the server are managed using a simple LRU scheme.

Data access: The simulation of the system discussed above is driven by two components, namely, data organization and data access. We assume, as stated earlier, that data is organized in segments and pages. In our model, there is one data segment and two corresponding index segments (one clustered, and one non-clustered).

Each transaction executed by a transaction manager accesses a mean of `TransSize` data pages. The set of pages accessed by a transaction is controlled by the spatial locality parameters, `HotBounds` and `ColdBounds`, and the degree of locality parameter, `HotAccess`. Whether a transaction uses an index to access the needed data pages (and hence, additionally access index pages, too) or directly accesses the data pages is dictated by the `NdxAccessProb` parameter. Furthermore, when a data page is modified (with `WritePercentage` probability) due to inserts, deletes, and updates, the modifications are eventually reflected in the index segments, too. The number of indexes that are updated is equi-probably chosen between zero and all the indexes. This is done in order to reflect updates of specific attributes which could be either indexed or non-indexed. Of course, inserts and deletes are reflected in all the indexes. When an index page is modified, the parent index page is also modified with a probability of `NdxParentUpd`.

Processing costs: For each index or data page that a transaction accesses, the simulation executes the RIC protocol in its entirety. The processing costs of the various activities in the simulation model are shown in Table 3. We distinguish between the cost of processing index pages (`NdxPageInst`) and data pages (`DataPageInst`). The other major processing costs in the system are the software overhead of messages and locking which have fixed and variable cost components in order to model the size dependent nature of these costs. We have to note here that locking has a variable cost component only in the case of the RIC protocol which is dependent on the size of the SCIS of the current index segment

Transactional assumptions: For the purposes of our experiments, we assume that all transactions execute at Isolation Level 4. In addition, we assume that the clients follow a *force-at-commit* policy which implies that the client sends back all the cached dirty pages to the server when a transaction commits. However, the server follows a *no-force* policy, *i.e.,* the server does not have to write back the dirty pages to disk at commit time [5]. Furthermore, we assume that index operations release physical locks acquired on index pages as soon as the operation is done using the page.

Parameter	HotCold	Unif	PvtUpd
TransSize	20	20	20
HotBounds	64 pgs	–	64 pgs
ColdBounds	Rest	DB	Rest
HotAccess	80%	0%	80%
HotWritePercent	10–70%	0	10–70%
ColdWritePercent	10–70%	10–70%	0%
NdxAccessProb	75%		
NdxParentUpd	10%		
Processing Costs			
DataPageInst	30,000 Instr		
NdxPageInst	15,000 Instr		
ScanRepos	15,000 Instr		
MsgCostFixed	20,000 Instr		
MsgCostVar	10,000 Instr/page		
LockCost	300 Instr		
LockCostVar	2 × (Index Seg Size)		

Table 3: Workload parameters and processing costs.

Write Percentage	Actual Write Access Percentage
10%	6%
20%	10.5%
30%	15%
40%	18%
50%	20%
60%	22.5%
70%	25%

Table 4: Relationship between `WritePercentage` and observed write accesses.

As mentioned earlier in Section 4.2, client transactions access data pages either through indexes or directly. Furthermore, for data pages accessed directly, the `WritePercentage` parameter is used to determine if the page is accessed for reading or writing (updates, inserts, or deletes). These modifying accesses could cause secondary, modifying accesses to index pages to reflect the changes made in the data pages. However, the secondary accesses are not a parameter of the simulation, but a result of the database configuration. Thus, the actual number of pages accessed by a transaction and the percentage of those access that are write accesses are different from the parameters specified for the simulation. Table 4 shows the percentage of total observed accesses that are write accesses for different `WritePercentage` values.

4.2.2 Results

In our experiments we used different workloads (HotCold, Uniform, and PrivateUpdate) to evaluate the performance of the RIC protocol and compare it with CBL. These workloads are based on the workloads described in [1, 2]. The experiments test the performance of the coherence protocols as the need for coherence changes by varying the percentage of write accesses to the database. In the following we discuss each workload and discuss the relevant results. Furthermore, we also compare the scalability of the two protocols by comparing their performance for a larger system and database size. The values for the simulation parameters for the various workloads can be found in Table 3.

HotCold workload: In the HotCold workload each node has associated hot and cold regions in the database. Most of the read and write accesses of a transaction (80%) running on a node go to the node's hot region, and the rest of the accesses (both, read and write) are uniformly distributed over the node's cold region. The cold region of a node could either be a hot region of another node or could be a region of the database which is cold for all the nodes.

Figure 4 shows the throughput of the two protocols for varying degrees of writes to the database. It should be clear from Figure 4 that the RIC protocol is able to handle the increasing degree of writes in a more efficient manner than CBL. This can be seen from the percentage difference between the throughput of the two proto-

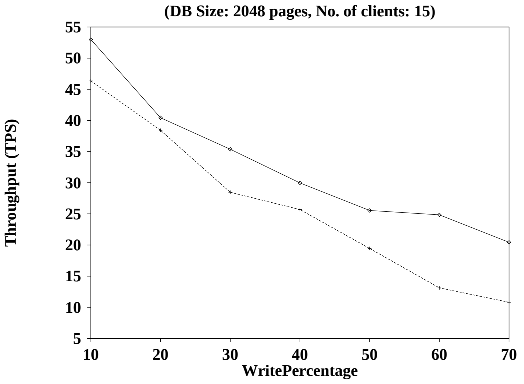

Figure 4: Throughput for HotCold workload.

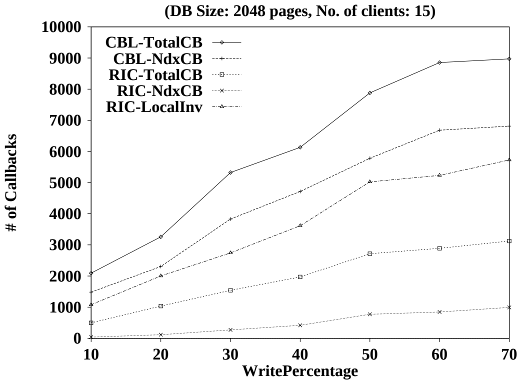

Figure 5: Coherence activity for HotCold workload.

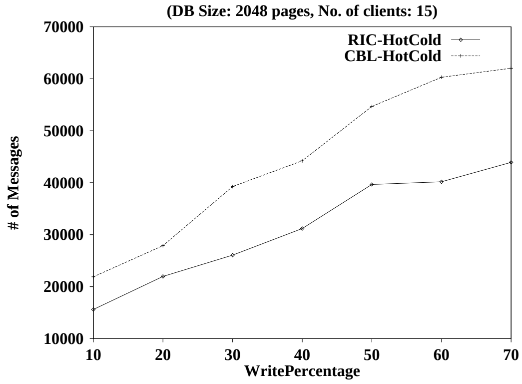

Figure 6: Messages exchanged for HotCold workload.

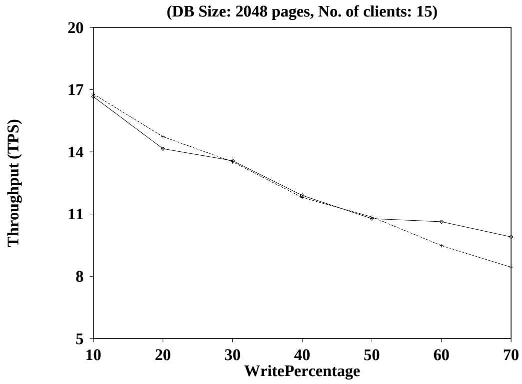

Figure 7: Throughput for Uniform workload.

cols which increases with the number of writes (about 14% for 10% `WritePercentage` to about 89% for 70% `WritePercentage`). The difference in performance between the two protocols is attributable only to the coherence activity required by each. Figure 5 shows the amount of coherence activity between nodes for which we use the number of callbacks issued as a metric. In the CBL protocol both the total number of callbacks and callbacks on index pages (lines labeled CBL-TotalCB, and CBL-NdxCB respectively) are very high. This is due to the fact that anytime that an index page has to be modified the server has to first callback all the cached copies of that page before the modifying client can proceed. In the RIC protocol, modification to index pages only result in callbacks if another node has the same index page cached for modification. Thus, the number of callbacks for index pages is fairly low and hence the total number of callbacks is also driven down. In Figure 5 we also show the number of local page invalidations (line labeled RIC-LocalInv) that happen when an index page in the cache is accessed and its validity is checked using the locally available SCIS. The growth of this number follows a similar pattern as the number of index callbacks (see line labeled CBL-NdxCB) in the case of CBL. This is to be expected because eventually both CBL and RIC have to invalidate roughly the same number of pages. The difference is that, CBL accomplishes this through messages whereas RIC accomplishes it locally using the coherence information obtained when data locks are acquired. Assuming that 20% of total accesses are write accesses (`WritePercentage` = 50%), the RIC protocol reduces the amount of coherence activity (number of callbacks) by as much as 65%.

In Figure 6 we show the total number of messages exchanged in the system for the HotCold workload for varying write percentages. The difference in messages exchanged in the RIC protocol and CBL protocol are significant (about 27% for `WritePercentage` = 50%). There are two reasons for this difference. First, the reduction in number of callbacks results in twice as large a reduction in the number of related messages because each callback involves two messages (request and grant). Furthermore, in RIC no physical locks are acquired for reading an index page which further reduces the number of messages.

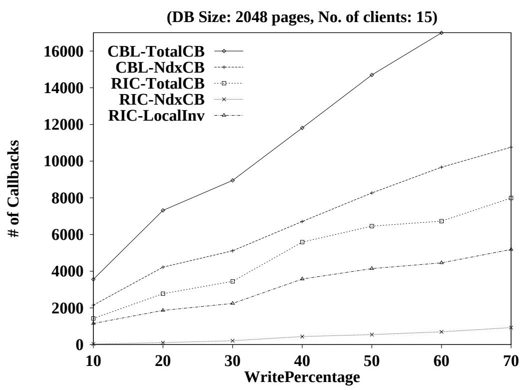

Figure 8: Coherence activity for Uniform workload.

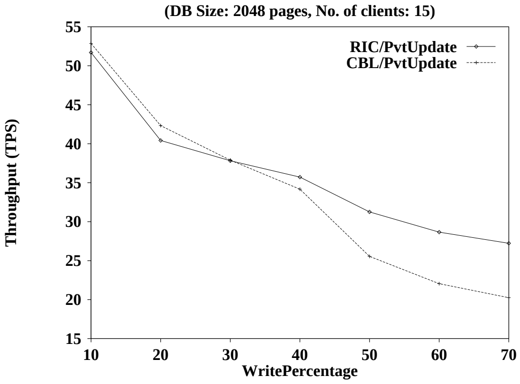

Figure 9: Throughput for PrivateUpdate workload.

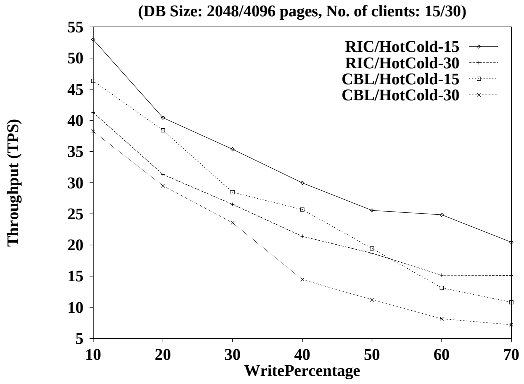

Figure 10: Throughput for different system sizes.

Uniform workload: In the Uniform workload, transactions access all the pages in the database with equal probability. Although, this workload may not portray any realistic Client-Server workload, it does stress test the coherence protocols.

Figure 7 shows the performance of the two protocols for the Uniform workload as a function of the `WritePercentage`. In this figure we notice that the performance gains that the RIC protocol provides are much smaller, varying between statistically insignificant to about 17% for 25% write accesses (`WritePercentage` = 70%). The reason for this is that, because of the workload access pattern the percentage of client-cache hits is significantly lower in the case of the Uniform workload (approximately 38%) when compared to the HotCold workload (approximately 72%). This behavior results in the addition of two costs: the cost of fetching pages from the server, and the cost of disk I/O (we observed that in the Uniform workload the number of disk I/Os were almost double that of the I/Os in the HotCold workload). Thus, even though the RIC protocol significantly reduces the coherence activity the cost of lack of locality in the workload overshadows the performance gains that it provides.

Figure 8 shows the amount of coherence activity in the system for the Uniform workload. Qualitatively this figure is very similar to Figure 5. The difference for a `WritePercentage` of 50% is about 56%. However, a closer comparison of Figures 8 and 5 reveals another subtle benefit of RIC, *i.e.,* the number of local invalidations in the Uniform workload (RIC-LocalInv) does not track the number of index callbacks (CBL-NdxCB) in the CBL protocol. This benefit is derived from the lazy invalidation of index pages that the RIC protocol performs only when index pages are actually accessed. The message performance of the two protocols in the Uniform workload is qualitatively very similar to the one presented in Figure 6.

PrivateUpdate workload: The PrivateUpdate workload is very similar to the HotCold workload in that it has both hot and cold regions and accesses go to these regions with different probabilities. However, in the PrivateUpdate workload, client transactions only modify pages in their hot regions and all accesses to cold data are read-only.

We notice from Figure 9 that the throughput of the two protocols is very similar for small write percentages (up to 15% of total accesses, or `WritePercentage` = 30%) but begins to diverge for larger write percentages. In the PrivateUpdate workload, all the writes are to a client's hot set. This implies that there are more local hits even for write accesses. However, writes to data pages also cause writes to index pages. Although the PrivateUpdate workload limits writes to the private hot pages of each node, the index pages (especially the non-clustered index pages) do not show the same amount of locality as the data pages because each index page can contain access paths to data pages from different hot/cold sets. So as the percentage of writes increases the frequency of modification of index pages which are cached at other nodes increases. Thus, for very low write percentages, where most of the index modifications are also localized, the RIC protocol in fact does a little worse than the CBL protocol (roughly 5%). The reason for RIC's performance degradation in this case is because the protocol uses the coarse grained segment-version numbers to determine if a SCIS has to be transmitted at a synchronization point, and hence transmits the version vector even though there might not be any inconsistency in the portion of the index that the requesting client is interested in. Furthermore, the CCM and the client transaction have to do additional processing before they can determine that the inconsistency detected is not relevant to the local operation. However, for an average percentage of about 20% (`WritePercentage` = 50%) the RIC protocol provides about 24% better throughput than the CBL protocol.

Scalability: One of the benefits of reducing the coherence activity (and messages) is improving the scalability of the system. From the results presented so far, we have seen that the RIC protocol reduces both the amount of coherence activity and the number of messages significantly when compared to the CBL protocol. In the following we examine if this improvement does help in the scalability of the system. First, we scale the size of the system in order to test the scalability of the two protocols. We scale the system by doubling both the number of clients and the size of the database. However, none of the server resources are scaled.

Figure 10 compares the throughput of the RIC and CBL protocols for the HotCold workload for system sizes of 15 and 30 clients, and database sizes of 2048 pages and 4096 pages, respectively. If the protocols were to support perfect scalability we would see identical curves for both system sizes. However, since the server is a bottleneck in this architecture, we do not see perfect scalability. We can see that both the RIC protocol and the CBL protocol support lesser transaction throughput for the larger system. However, we can also see that the percentage degradation of the throughput is lower in the case of the RIC protocol (lines labeled RIC/HotCold-15 and RIC/HotCold-30) than the CBL protocol (lines labeled CBL/HotCold-15 and CBL/HotCold-30). For example, for about 20% write accesses (`WritePercentage` = 50%) the percentage degradation in the throughput of the RIC protocol is about 27% whereas the throughput of the CBL protocol is down by about 42%. Furthermore, the throughput of the RIC protocol for the smaller system is about 35% higher than that of the CBL protocol, whereas for the larger system it is higher by about 66% indicating that the performance of the RIC protocol is degrading at a slower rate than the CBL protocol. These results attest to the better scalability of the system using the RIC protocol.

5 Concluding Remarks

One of the main impediments to transaction throughput and scalability in a client-server database environment is the overhead in maintaining the coherence of the data and index pages shared by the clients. In this paper, we identified design guidelines from the point of view of achieving increased transaction throughput as well as scalability, namely, reducing the amount of explicit client-server communication for page/lock requests, and reducing the amount of implicit communication in the form of coherence activity. We presented a new protocol, called Relaxed Index Consistency, that exploits the difference in access semantics between data and index pages. This protocol relies on the fact that it is possible to tolerate a certain amount of inconsistency in the index pages during the positioning phase of index operations. Using data-only locking, the proposed protocol restores consistency for index pages during the operation phase of the index operations. We have shown qualitatively and through simulation studies the performance benefits of the RIC protocol both in terms of throughput and scalability.

In this paper, we have not considered the recovery implications of RIC. However, we expect that recovery protocols, such as those described in [5, 18], would be adequate because all writes under RIC are serialized and no transactions are aborted or rolled back due to inconsistent index accesses. In the RIC protocol, the unit of index coherence is a segment, and in this paper we have only considered segments that represent whole indexes. However, the segment size can be made smaller (say, a sub-tree in the index) in order to reduce the size of the coherence messages (for very large indexes), and to allow more precise coherence-control, as was required in the PrivateUpdate workload. In our future work we would also like to explore the usage of multiple-writer semantics, and fine-granularity locking, as suggested in [2], in the RIC protocol, and their impact on the throughput and scalability of the system.

[6] contains a longer version of this paper.

References

[1] M. J. Carey, M. J. Franklin, M. Livny, and E. J. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In *Proceedings of the 1991 ACM SIGMOD Conference*, pages 357–366, 1991.

[2] M.J. Carey, M.J. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proceedings of the 1994 ACM SIGMOD Conference*, pages 359–370, May 1994.

[3] D. M. Dias, B. R. Iyer, J. T. Robinson, and P. S. Yu. Integrated Concurrency–Coherency Controls for Multisystem Data Sharing. *IEEE Transaction on Software Engineering*, 15(4):437–448, April 1989.

[4] M. Franklin and M. Carey. Client-Server Caching Revisited. In *Proceedings of the International Workshop on Distributed Object Management*, August 1992.

[5] M. Franklin, M. Zwilling, C. K. Tan, M. Carey, and D. DeWitt. Crash Recovery in Client-Server EXODUS. In *Proceeding of the 1992 ACM SIGMOD Conference*, pages 165–174, June 1992.

[6] V. Gottemukkala, E. Omiecinski, and U. Ramachandran. Relaxed Index Consistency for a Client-Server Database. Technical Report GIT-CC-95-33, College of Computing, Georgia Institute of Technology, Atlanta, GA, 1995.

[7] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[8] J. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[9] T. Johnson and A. Colbrook. A Distributed Data-balanced Dictionary based on the B-link Tree. In *Proceedings of the International Parallel Processing Symposium*, pages 319–325, 1992.

[10] T. Johnson and P. Krishna. Lazy Updates for Distributed Search Structure. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 337–346, 1993.

[11] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[12] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993.

[13] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operation on B-Tree Indexes. In *Proceedings of the 16th VLDB Conference*, August 1990.

[14] C. Mohan. Interactions Between Query Optimization and Concurrency Control. In *Proceedings of the International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 26–35, February 1992.

[15] C. Mohan. Less Optimism About Optimistic Concurrency Control. In P. S. Yu, editor, *Proceedings of the International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 199–204. IEEE Computer Society Press, February 1992.

[16] C. Mohan and I. Narang. Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment. In *Proceedings of the 17th VLDB Conference*, September 1991.

[17] C. Mohan and I. Narang. Efficient Locking and Caching of Data in the Multisystem Shared Disks Transactions Environment. In *Proceedings of the 3rd International Conference on Extending Database Technology*, pages 453–468, 1992.

[18] C. Mohan and I. Narang. ARIES/CSA: A Method for Database Recovery in Client-Server Architectures. In *Proceedings of the 1994 SIGMOD Conference*, pages 55–66, May 1994.

[19] H. Schwetman. *CSIM Users' Guide*, March 1990.

[20] Y. Wang and L. A. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *Proceedings of the 1991 ACM SIGMOD Conference*, 1991.

[21] W. E. Weihl and P. Wang. Multi-Version Memory: Software Cache Management for Concurrent B-Trees. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 650–655, 1990.

[22] K. Wilkinson and M-A. Neimat. Maintaining Consistency of Client-Cached Data. In *Proceedings of the 16th VLDB Conference*, pages 122–133, 1990.