Performance Modeling of an Enhanced Optimistic Locking Architecture for Concurrency Control in a Distributed Database System

A.A. Akintola, G.A. Aderounmu and A.U. Osakwe

Department of Computer Science and Engineering
Obafemi Awolowo University, Ile-Ife, Nigeria

M.O. Adigun

Department of Computer Science
University of Zululand, Kwadlangezwa, Republic of South Africa
{aakintola69, gaderoun}@yahoo.com; alukay2k@hotmail.com; madigun@pan.uzulu.ac.za

Optimistic methods of concurrency control are gaining popularity. This is especially true with the resurgence of mobile and distributed databases during the last decade, which inherently rely on optimistic techniques to improve availability and performance of the distributed database systems. A key problem with optimistic techniques is that they do not perform well in highly conflict prone environments. Pessimistic techniques, especially locking, perform much better under these circumstances. In this research effort, the authors explored a hybrid technique called optimistic locking architecture that provides locking for high conflict data items and optimistic access for the rest. The model uses self-tuning that does not require the transaction manager, the transaction or the user to incorporate any additional knowledge or to specify which data items or transactions are optimistic. Rather, the system uses a data structure called lock buffer to maintain optimal level of locks in the system. The empirical results obtained show that the performance of optimistic concurrency control techniques can be significantly improved by using a relatively small lock buffer. The analysis of the results also show improved performance degradation which is suitable to the distributed database environment.

ACM Classification: C.2.4: Distributed Systems

1. INTRODUCTION

Concurrency control in a database system is the activity of coordinating the actions of transactions that operate in parallel; access shared data, and potentially interferes with one another. It is desirable that this coordination be efficient. There are two costs associated with concurrency control: Lost opportunity cost and restart cost. The former cost is a significant factor in conservative methods, which involve waiting to ensure that there will be no conflict or interference. Some of this waiting may be unnecessary, constituting a lost opportunity cost. Restart cost is significant in aggressive methods, which optimistically execute transactions, based on the assumption conflict does arise; some transactions must be aborted and restarted, thus incurring a restart cost. Devising a concurrency

Manuscript received: 16 October 2004
Communicating Editor: Arthur Sale

control technique that has both a low lost opportunity cost and a low restart cost is a hard problem (Rashmi, 2002).

With the demand for higher performance and higher availability, computers have moved from centralized to distributed architectures. This introduces new issues in the area of database management. In a distributed database systems, according to Imielinski and Badrinath (1994), the data is stored at a number of sites that are geographically distributed over a possibly large region, a country or even the whole world. For many distributed applications like banking, telecommunications, etc. distributed database represent a more natural and appropriate solution than monolithic, centralized systems. Many of today's commercial database system such as Oracle 9i or IBM DB2 Propagator provide the required support for data distribution and inter-database communication. As new communication technologies are emerging, wireless and mobile computing concepts become reality and allow for an even higher degree of distributed databases. Wireless technology thus expands the scope of distributed applications by enabling ubiquitous database interaction anywhere and anytime.

In this evolving world of distributed databases, data replication plays an increasingly important role. Replication intends to increase data availability in the presence of site or communication failures, and to decrease retrieval costs by local access if possible. The maintenance of replicated data is therefore closely related to intersite communication, and replication management can have significant impact on the overall system performance (Nicola and Jarke, 2000). During the last years, several distributed systems have been realized. Usually, the concurrency control in these systems has been done by some kind of two-phase locking, but as processor speed increases relative to I/O and communication speed, it is expected that timestamp ordering should be able to compete with two-phase locking in performance (Norvag, Sandsta and Bratbergsengen, 1997).

While the distributed concurrency control performance studies to data have been informative, a number of important questions remain unanswered. These include:

- how do the performance characteristics of the various basic algorithm classes compare under alternative assumptions about the nature of the database, the workload, and the computational environment?
- how does the distributed nature of transactions affect the behaviour of the various classes of concurrency control algorithms?
- how much of a performance penalty must be incurred for synchronization and updates when the data is replicated for availability or query performance reasons?

According to Kohler and Jeng (1985), most distributed concurrency control algorithms fall into one of three basic classes: locking algorithms, timestamp algorithms, and optimistic or certification algorithms. In this paper, the authors intend to use a hybrid model comprising of the locking and optimistic algorithms hence the name of the proposed architecture-optimistic looking model.

2. DISTRIBUTED DATABASE SYSTEM

A distributed database system (DDBS) is a collection of several logically related databases which are physically distributed in different computers (otherwise called sites) over a computer network (Ozsu and Valduriez, 1991; Bell and Grimson, 1992). All sites participating in the distributed database enjoy local autonomy in the sense that the database at each site has full control over itself in terms of managing the data. Also, the sites can inter-operate whenever required. According to Date (1991), the user of a distributed database has the impression that the whole database is local except for the possible communication delays between the sites. This is because a distributed database is a logical union of all the sites and the distribution is hidden from the user.

DDBS is preferred over a non-distributed or central database system (DBS) for various reasons. Distribution is quite common in an enterprise. For instance, various branches of the enterprise can be located at various geographic locations. Also, there can be logical divisions in a branch such as human resources department and administration department. Each logical division can be treated as a site. At each site, a DDBS provides all capabilities of a DBS local to that site in addition to providing other advantages of a distributed system such as replication and fault tolerance. Though the concept of data distribution seems to be attractive, the data is very high. Also, data distribution makes concurrency control and recovery mechanisms complex. In this research effort, the authors intend to look at concurrency control mechanisms and how a hybrid scheme can be deployed effectively.

In a DDBS, the presence of distribution complicates the coordination mechanism of sites. Three important issues differentiate the DDBS from the traditional DBS, these are:

- Replication – this can be done by storing multiple copies of an entire table or partitions of a table (caked fragment) at multiple sites (Elmasri and Navathe, 2000). According to Date (1991), replication provides at least two benefits: high availability and improved performance.
- Fault tolerance – the problems due to distribution become complex during site failure (s). Failures can occur at various levels of a DDBS. Bell *et al* (1992) categorized failures in a DDBS under four headings: transaction – local failures, site failures, media failures, and network failures.
- Concurrency control- in DDBS, the concept of transaction is complicated because a transaction may access data stored at more than one site. Typically, a transaction is divided into sub-transactions that can be executed at different sites. Bell *et al* (1992) defines transaction as a series of actions carried out by a user or application. The actions defined in a transaction are either executed completely or not executed at all (this property is also termed atomicity). The uncertainty in the order of execution of sub-transactions may result in different output or different state of the database. In this context, concurrency control is a process of controlling the relative order of two or more sub-transactions that interfere with each other.

3. ARCHITECTURE OF DISTRIBUTED DATABASE

A distributed database management system (DDBMS) involves a collection of sites interconnected by a network. Each site runs one or more of the following software modules: a transaction manager (TM), a data management (DM), and a concurrency control scheduler (or simply scheduler). In a client-server model, a site can function as a client, a server, or both. A client runs only the TM module, and a server runs only the DM and scheduler modules. Each server stores a portion of the database. Each data item may be stored at any server or redundantly at several servers. Figure 1 shows the system architecture for the client-server model. Users interact with the DDBMS by executing transactions, which are on-line

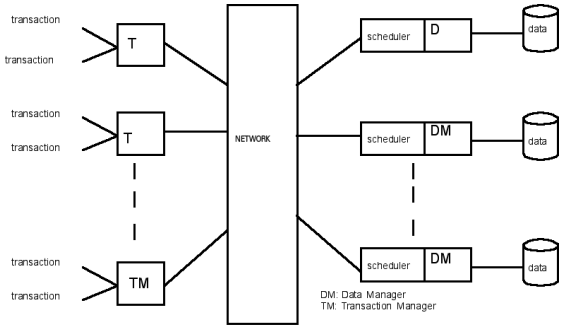


Figure 1: Architecture of a Distributed Database System

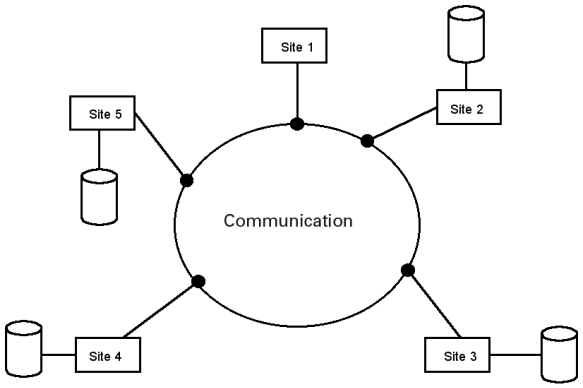


Figure 2: Distributed Database Environment

queries or application programs. TMs supervise interactions between transactions and the database. The TM at the site where the transaction originates is called the initiating TM. The indicating TM receives operations issued by a transaction, and forwards them to the appropriate schedulers. The goal of a scheduler is to order operations so that the resulting execution is correct. DMS manage the actual database by executing operations, and are responsible for recovery from failures. Transactions communicate with TMs, TMs communicate with schedulers and DMs, and DMS manage data.

Architecturally, a DDBS consists of a (possibly empty) set of query sites and a non-empty set of data sites. The data sites have data storage capability while the query sites do not. The latter only run the user interface (in addition to applications) in order to facilitate data access at data sites, as shown in Figure 2. According to Ozsu *et al* (1991), the problem of distributed query processing is to decide on a strategy for executing each query over the network in the most cost- effective way. Two measures of query optimization are response time and throughput. Response time is the time taken by a system to respond to a query. Throughput is the average number of transactions successfully passing through the system. The problem is non-deterministic polynomial-hard (NP-hard) in nature, and the factors to be considered are distribution of data, communication costs, local processing costs, etc. The authors believe that concurrency control and recovery are two important transaction managements.

3.1 Distributed Transaction Management

In a DDBMS, the challenge in synchronizing concurrent user transactions is to extend both the serializability argument and the concurrency control algorithms to the distributed environment. The global serializability requires that the execution of the set of transactions at each site is serializable and that the serialization orders of these transactions at all these sites are identify\cal. A distributed DMBS also has to deal with communication failures. Distributed recovery protocol deal with the problem of recovering the database at a failed site to a consistent state when that site recovers from the failure. A general distributed transaction in terms of the processes involved in its execution is depicted in Figure 3. Each transaction has a master process (M) that runs at its site of origination. The master process in turn set up a collection of cohort processes (Ci) to perform the actual processing involved in running the transaction. Since virtually all query processing strategies for DDBSs involve accessing data at the site(s) where it resides, rather than accessing it remotely, there is at least one such cohort for each site where data is accessed by the transaction. DDBSs differ in the degree of parallelism involved in query execution. In general, data may be replicated, in which case each cohort that updates any data item is assumed to have one or more update (Uij) processes associated with it at other sites.

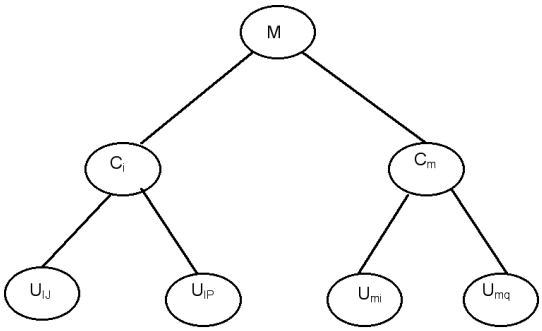


Figure 3: Distributed Transaction Structure

In particular, a cohort will have an update process at each remote site that stores a copy of the data items that it updates. It communicates with its update processes for concurrency control purposes, and it also sends them copies of the relevant updates during the first phase of the commit protocol. When a cohort finishes its portion of a query, it sends an execution complete message to the master. When the master has received such a message from each cohort, it will initiate the commit protocol by sending prepare to commit protocol by sending prepare to commit messages to all sites. Assuming that a cohort wishes to commit, it sends a 'prepare' message back to the master, and the master will send commit messages to each cohort after receiving prepared messages from all cohorts. The protocol ends with the master receiving 'committed' messages from each of the cohorts. If any cohorts are unable to commit it will return a cannot commit message instead of a 'prepared' message in the first phase causing the master to send 'abort' instead of 'commit' messages in the second phase of the protocol. When replica update processes are present, the commit protocol becomes a nested two-phase commit protocol.

4. RELATED RESEARCH

Several concurrency control techniques have been proposed, based on different types of schedulers. A scheduler can be conservative or aggressive or a combination of the two. A conservative or a combination of the two. A conservative scheduler delays operation until it is certain that there will be no conflicting operation that is due to arrive. Therefore, conservative techniques suffer from a lost opportunity cost. An aggressive scheduler reduces unnecessary delay by scheduling operations immediately. However, it runs the risk of having to reject operations later, thereby causing the issuing transaction to abort and restart.

Traditionally, concurrency control techniques have been classified into four categories – locking, timestamp ordering, optimistic, and hybrid (Rashmi, 2002; Thomasian, 1998; Yu, Wu, Lin and Son, 1994). In this paper, we will consider some of the schemes under each of the categories and their associated problems. Concurrency control techniques in distributed databases have been analytically studied, but these analytical studies make restrictive assumptions (Gray and Reuter, 1992). Existing analytical models for Two-Phase Locking (2PL) in a distributed database do not permit simultaneous processing of a transaction at multiple sites (Gray *et al*, 1992). Basic Timestamp ordering techniques have also been analyzed in Sheikh and Woodside (1997). It was reported in Carey and Linvy (1991) that BTO performed worse than 2PL in a distributed database, especially in a low data contention scenario. In the case of Wound-Wait (WW), deadlocks are prevented via the use of timestamps. According to Rashmi (2002), each transaction is numbered according to its initial startup time, and younger transactions are prevented from making older transactions wait. If an older transaction

requests a lock, and if the request would lead to the older transaction waiting for a younger transaction, the younger one is 'wounded' – it is restarted unless it is already in the second phase of its commit protocol. A major drawback of WW avoidance scheme is that the request for a data item held by another transaction does not imply a deadlock. It therefore incurs the associated message costs. Also, the abort and rollback may not be necessary. In the case of Distributed Certificate (OPT), it always uses restarts to handle conflicts, checking for problems only when a transaction is ready to commit. This means that it defers communication between cohorts and updates until commit time. Piggybacking its concurrency control information on the message of the commit protocol can be lethal. Finally, few of the available hybrid schemes for concurrency control are discussed (Rashmi, 2002; Halici and Dogac, 1991; Yu and Dias, 1992). A distributed optimistic 2PL scheduler executes transactions optimistically, but if a transaction is aborted, the scheduler uses 2PL to execute the transaction a second time. We also have optimistic with dummy locks, hybrid optimistic concurrency control and broadcast optimistic concurrency control. All these hybrid schemes are not suitable for high conflict data items. Even in Halici *et al* (1991), the concurrency control protocol is locally pessimistic and uses locking. However, actual propagation of locks and updates to sites not participating in the execution of the transaction is delayed until the transaction commits. Thus, the commits take place in an optimistic fashion; with inter-site read/write conflicts detected only at commit time. However, the model in this research effort is different in that there is no partitioning of transactions into pessimistic or optimistic classes; rather each transaction is potentially a hybrid transaction. Furthermore, the transactions do not control the set of data items, which are optimistic.

5. SYSTEM MODEL

In this section, a hybrid technique that provides locking for high conflict data items and optimistic access for the rest is explored. While hybrid techniques have been proposed in earlier literature, especially in Halici *et al* (1991), this technique is unique in that it is self-tuning and does not require the transaction manager, the transaction or the user to incorporate any additional knowledge or to specify which data items or transactions are optimistic. Rather, the system uses a data structure called *lock buffer* to maintain an optimal level of locks in the system. This data structure enhances the performance of the basic optimistic model by automatically providing locking for highly conflict prone data. A unique feature of the algorithm formulated in this model, is that locks may be evicted from the lock buffer, if the number of data items for which locks are requested exceeds the size of the lock buffer. All transactions affected by such an eviction of locks automatically 'become' optimistic with respect to the evicted data items.

The design of the lock manager is the key to this hybrid technique. The lock manager maintains a finite lock buffer. Each slot (or frame) in the lock buffer holds locks and pending lock requests for a single data item. Thus, the number of data items with active lock requests cannot exceed the number of slots in the buffer. Whenever a lock request for a data item x is received by the lock manager, the lock manager first attempts to locate x in the lock buffer. If it is found the lock manager attempts to post the lock in the corresponding slot. The lock request can either be granted or be blocked in the same fashion as pure locking depending on the status of existing locks on x. Thus, a read (shared) lock would be granted if and only if there is no existing exclusive lock on x. A write (exclusive) lock would be granted if and only if there are no existing locks on x. If the lock is granted the transaction now holds or owns a lock on x. If x is not located in the lock buffer, a slot must be located for posting the lock request. If a free slot exists, it will be used; otherwise a victim slot must be selected. Note that any slot with no locks or pending lock requests is considered free.

If the size of the lock buffer is zero, then all the lock requests get rejected and there are never any

active locks. In this scenario all transactions become optimistic with respect to all the data items in their read and write sets, and the system becomes purely optimistic. On the other hand, if the number of slots is greater than or equal to the number of data items in the database, then each lock request can be granted without evicting any existing data items and locks.

On commit, the transaction must be validated. For a transaction to be valid, it must be valid with respect to all of the data items in its read and write sets. A transaction T is valid with respect to a data item x if one of the following three conditions hold (a concurrent transaction T' to T is one whose execution overlaps T; i.e. either T' started after T but before T entered its validate phase or T started after T' but before T' entered its validate phase):

1. T holds a lock on x (i.e. the lock acquired by T on x was not evicted).
2. x is not in the writeset of T, T does not hold a lock on x and both of the following conditions hold:
   a) if another transaction T' holds a lock on x then it is not an exclusive lock
   b) x was not overwritten after it was read by T (i.e. x is not in the writeset of any concurrent committed transaction).
3. x is in the writeset of T, T does not hold a lock on x and both of the following conditions hold:
   a) No other transaction holds a lock on x (either shared or exclusive).
   b) x was not overwritten after it was "read" by T (i.e. x is not in the writeset of any concurrent committed transaction).

After validation succeeds, the transaction's writeset is written to the database and all locks held by the transaction are released. These conditions are incorporated into Algorithm 1, which is the validation algorithm. For a transaction T, READSET(T) is the set of data items read by T, WRITESET(T) is the set of data items written by T and LOCKSET(T) is the set of all data items that T owns locks for.

```
Ensure: // Inputs: Transaction T
1:  // Main Loop:
2:  for all x ∈ READSET(T) do
3:      if x ∉ LOCKSET(T) then
4:          if x ∉ WRITESET(T) Λ ∃T' : x ∈ WLOCKSET(T') then
5:              Abort T
6:              return
7:          end if
8:          if x ∈ WRITESET(T) Λ ∃T' : x ∈ LOCKSET(T') then
9:              Abort T
10:         return
11:     end if
12:     if ∃committed T' concurrent to T: x ∈ WRITESET (T') the
13:             Abort T
14:             return
15:     end if
16:   end if
17: end for
18: Commit T
19: END
```

Algorithm 1: Validation Algorithm

5.1 Components of the Model

We model a distributed database as a collection of data items (tuples). A tuple can be used to represent a relation, or it can represent a partition of a relation in a system where relations are partitioned across multiple sites. Tuples are assumed to be the unit of data replication. Figure 4 shows the structure of our model while Figure 5 shows the sub-systems of the model. Table 1 summarizes the parameters of the database model, which include the number of sites and tuples in the database and the sizes of the files. The mapping of tuples to sites is specified via the parameter tuple locations, a boolean array in which $\text{TupleLocations}_{ij}$ is true if a copy of data items i resides at site j. The basic components of the model are discussed below:

- Transaction Manager: this module provides all the functionality of a transaction manager for each transaction, including handling of lock requests, page requests, deadlock management, and the read, validation and write phases of transactions. These functions are executed by the CPUs. Deadlocks are managed using deadlock avoidance. The transaction manager is responsible for accepting transactions and modeling their execution. To choose the execution sites for a transaction's cohorts, the decision rule is: If a tuple is present at the originating site, use the copy there; otherwise, choose uniformly from among the sites that have remote copies of the tuple. If the tuple is replicated, the transaction manager will initiate updaters at sites of other copies when the cohort accessing the file first needs to interact with them for concurrency control reasons. The transaction manager also models the details of the commit and abort protocols.
- Concurrency Control Manager: This module manages the *Lock Buffer*, handles lock and unlocks requests along with functionality to evict data items from the lock buffer, and performs validation phase of transactions.

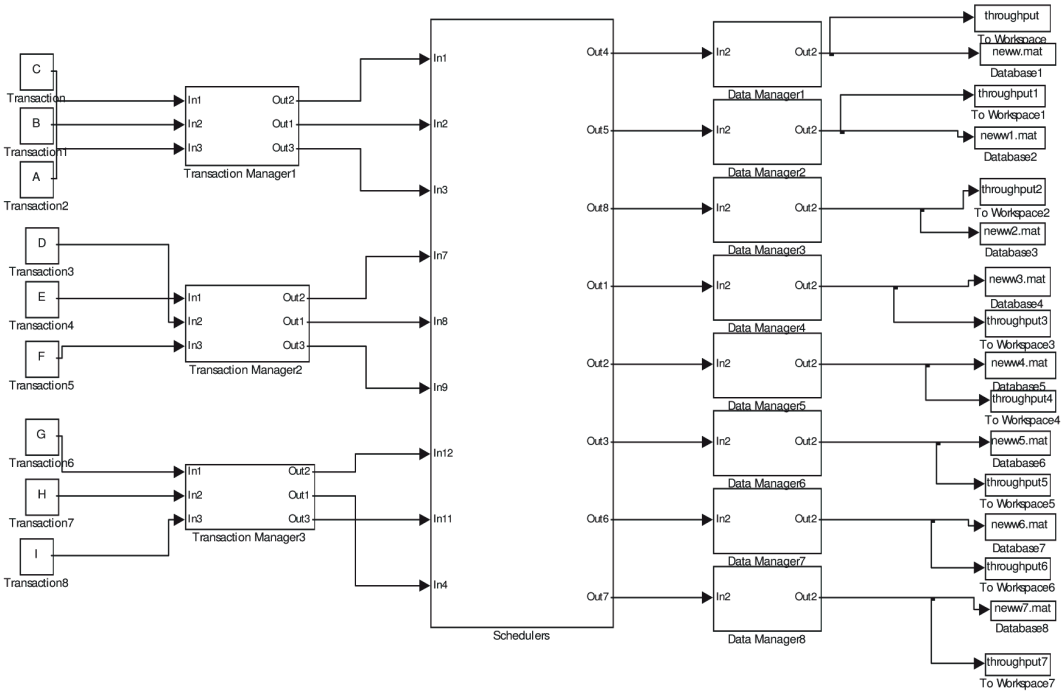


Figure 4: Optimistic Locking Model Structure

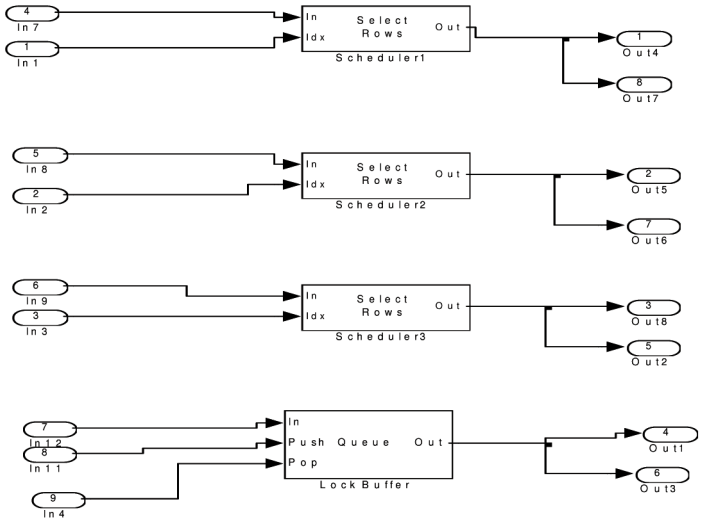


Figure 5: Sub-systems of the Optimistic Locking Model

Parameter	Description	Value
NumSites	Number of sites in the DDBS	8 sites
NumTuples	Number of tuples in the relation.	100000 per site
TuplesPerPage	Number of tuples in a page	10 pages per tuple
NumDisks	Number of disks	10 per site
BufferSize	Number of frames in the buffer pool	1000 per site
QueueLen	Minimum length of the pending queue	0.1 – 1.0
ProbWrite	Probability of a transaction being read-write	0.1 – 1.0
TransactionSize	Average number of tuples accessed by a transaction	1000 – 10000
ProbReqWrite	Probability of a tuple accessed by a read-write transaction being a write	0.1 – 1.0
NumCPU	Number of CPUs	10 CPUs per site
DegMulti	Maximum degree of multitasking on each CPU	10
LockBufferSize	Size of the lock buffer	0 – 100000
TimePerTuple	Time for processing a tuple	10 milliseconds

Table 1: Parameters used in the Model and their values

- Data Manager: The DM executes each read and write operation it receives. The data manager simulates the physical database storage manager. Each node has a data manager, which operates independently of the other data managers in the system. We have used a very simple model, where the data manager only consists of a simulated buffer cache and a disk. Time delays are used to simulate operations against the data manager. The time needed to execute a disk operation is the sum of the time that operation has to wait for the disk to become idle (the time needed to execute previous operations submitted to the disk), and the time it takes to read or write the page. We have chosen a very simple algorithm for disk scheduling: *first-come-first-served* (or *first-in-first-out*). A read operation from the cache takes constant time, and can be served immediately.

5.2 Correctness

The correctness of Algorithm 1 is sketched in this section. The full proof requires a consideration of all lock/operation conflicts. The notation used by Bernstein, Hadzilacos and Goodman (1987) is also used here.

To prove correctness of this algorithm, there is the need to show that all histories produced by the algorithm are serializable. Suppose H is a history produced by the algorithm, and SG(H) is the corresponding serializability graph. By the serializability theorem (Bernstein *et al*, 1987), H is serializable if and only if SG(H) is acyclic.

Suppose T_i and T_j are transactions in H and $T_i \rightarrow T_j$ is an edge in SG(H). Thus there must exist a pair of conflicting operations on some data item x, p_i (x) in T_i and q_j (x) in T_j such that $p_i(x) < q_j(x)$ (i.e. p_i (x) occurred before q_j (x)). For the operations to be conflicting, either p_i (x) or q_j (x) or both must be writes. We claim that T_i must have committed before T_j (note that commits are atomic). Suppose instead that T_j committed before T_i. There are two (non mutually exclusive) cases:

1. q_j (x) is a write. For T_j to commit, T_i may not own a lock on x. If it does then by applying rule 2a or 3a, we must abort T_j, contradicting our assumption that T_j committed. Now suppose T_j commits writing x. Then we must abort T_i by either one of rule 2b or 3b since the data item x was overwritten since it was accessed by T_i (note that writes are always treated as reads followed by writes in our model). This is also a contradiction.
2. p_i (x) is a write. The only interesting case here is if T_i owns an exclusive lock on x. If not, the write to x is not scheduled until T_i commits, which implies that q_j (x) < p_i (x) (since T_j committed before T_i) contradicting our assumption. However, if T_i owns an exclusive lock on x (which in turn implies that T_j does not own a lock on x), then by rule 2a or 3a, T_j must be aborted, contradicting our assumptions.

Thus, T_i must commit before T_j. Then, if there is a cycle in SG(H) each transaction in the cycle committed before itself, which is a contradiction. Thus, SG(H) is acyclic and H is serializable, as required.

5.3 Operations of the Model

The distributed database is a single relation consisting of NumTuples tuples. The relation is stored on disk as a set of pages, each page consisting of TuplesPerPage tuples. The pages in turn are striped across NumDisks disks. All page accesses go through the Data manager, which manages writeback buffer cache consisting of BufferSize slots. Each slot can hold a single page.

Transactions are generated at maximum rate as follows: a "pending" queue of transactions waiting to be scheduled is maintained by the transaction manager, and every time the number of transactions in this queue drops below a threshold QueueLen, new transactions are created. Thus the system receives transactions at the maximum possible rate.

A set of tuples in the form of I/O requests is generated for each transaction. A transaction is readwrite (i.e. an update transaction) with the probability ProbWrite, otherwise it is read-only (i.e. it is a query transaction). The number of requests is uniformly distributed with an average of TransactionSize. For a read-write transaction a request is a write with probability ProbReqWrite.

The transactions are sent to the transaction manager after being created. The transaction manager attempts to schedule the transaction on one of the NumCPUs CPUs in the system. Each CPU can multi-task at most DegMulti transactions (including blocked transactions). Transactions that cannot be scheduled are held in a pending queue awaiting a free CPU.

On being scheduled, a transaction starts its read phase. In this phase, locks are acquired for each

I/O request. However, only the read requests are actually passed to the buffer manager, while all write requests are held pending the write phase. Locking is performed in a recoverable 2-phase manner. Thus, all locks are acquired right before the requests are scheduled, but are not released till the end of the write phase.

At the end of the read phase, the transaction enters an atomic validation phase. For each tuple in the readset and writeset of the transaction the concurrency control manager checks if one of the conditions in Section 5 holds. If the transaction is valid with respect to all the tuples it accessed, the transaction enters a write phase where all pending writes are written to disk.

All the locks are managed by the concurrency control manager using a lock buffer of LockBufferSize tuples. The concurrency control manager is also responsible for blocking transactions whenever locks cannot be granted, and for unblocking them whenever either the lock is granted or the corresponding tuple is evicted from the lock buffer.

6. SYSTEM SIMULATION AND RESULTS ANALYSIS

In this section, we carried out a simulation of the developed model using MATLAB 6.5 and SIMULINK 4.1 with the following simulation parameters:

- **TransactionSize** ($|T|$): TransactionSize is the average number of tuples accessed by a transaction. The value of the TransactionSize was varied between 0 and 5000.
- **Lock Buffer Size**: The lock buffer size is the size of the lock buffer. The value of the Lock Buffer Size is varied between 0 and 10000.
- **Time per Tuple**: Time for processing each tuple. This value is measured in 10milliseconds.
- **Probability of Write** ($\mathbf{P_t}$) :This is the probability of a transaction being read-write. This value is varied between 0.1 and 0.5
- **Probability of Required Write** ($\mathbf{P_w}$): Probability of a tuple accessed by a read-write transaction being a write. This value is varied between 0.1, 0.5 and 1.0.

6.1 Analysis of Simulation Parameters

The values of all the parameters that were used for the simulation are given in Table 1. The simulation was conducted on two broad classes of runs. For the first class, TransactionSize was set to 1000 (i.e. 1% of the database size) and ProbWrite was set to 0:1 and 0:5 (i.e. 10% and 50% of the transactions were read-write), and ProbReqWrite was set to 0:1, 0:5 and 1:0 (i.e. 10%, 50% and 100% of the tuples accessed by the read write transactions were written to). This gives us 6 sets of results. For the second class of runs, both ProbWrite and ProbReqWrite are fixed to 0:1 and TransactionSize varied from 1000 to 5000 (i.e. between 1% and 5% of the database size). This gives another 2 sets of results (one each for 1000, 2500, 5000). For the results presented here the transactions were generated at maximum rate as described earlier. The experiments were conducted with different transaction inter-arrival rates, which have not been presented here.

The experiment was conducted using a series of runs with different lock buffer sizes. Significant results are summarized in Figures 6 through 13. In each case the independent parameter is the lock buffer size scaled down by a factor of 1000. These results are subject to a maximum variation of less than 2:3%. For the purpose of graphing, we denote ProbWriteby P_t, ProbReqWriteby P_w and TransactionSize by $|T|$.

6.2 Analysis of Results

In this section, the authors present the analysis of the results obtained from the simulation. The following observations were discovered:

6.2.1 Degree of Pessimism/Optimism of the System

Figures 12 and 13 are crucial for determining the degree of pessimism in the system. The slot eviction rate as well as the fraction of locks rejected is essentially measuring the same aspect of the system: how many data items 'became' optimistic. The slot eviction rate measures the rate at which data items become optimistic, whereas the fraction of locks rejected measures the absolute magnitude of locks rejected. Note that eviction of a slot may result in eviction of one or more locks. This could happen either if there are blocked lock requests or multiple granted locks. Thus the slot eviction rate is not a good measure of the absolute magnitude of the locks rejected.

As the number of locks rejected approaches one, the system becomes more and more optimistic. At unity, the system is fully optimistic. Furthermore, as the fraction of locks rejected approaches zero, the system becomes more and more pessimistic. When this number becomes zero, the system is fully pessimistic and follows two phase locking (2PL). It might appear in the graphs that lock eviction rate reaches zero before the lock rejection rate. This is merely an artifact of the precision of the graphs.

In each of these graphs, the system becomes fully pessimistic even though the size of the lock buffer is less than the size of the database. After this point, increasing the size of the lock buffer does

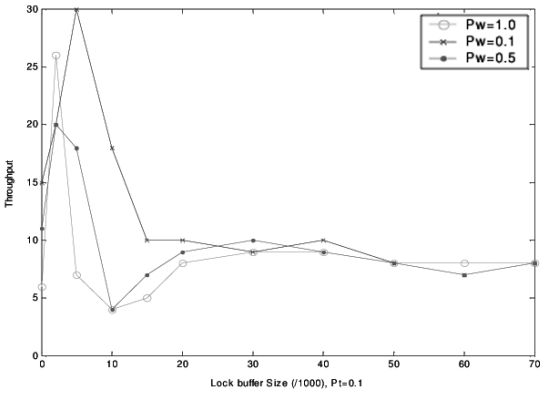


Figure 6: Throughput (Completed Transactions/Second) Pt = 0.1, $|T| = 1000$

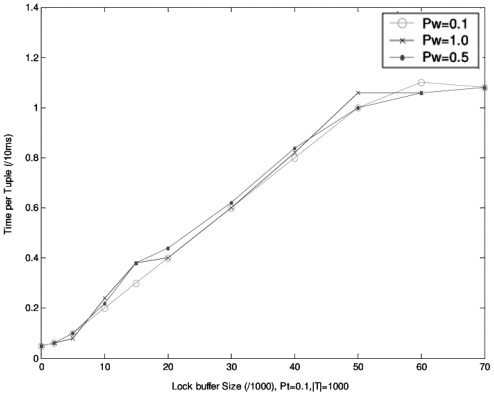


Figure 7: Time per Tuple (/10mns), Pt = 0.1, $|T| = 1000$

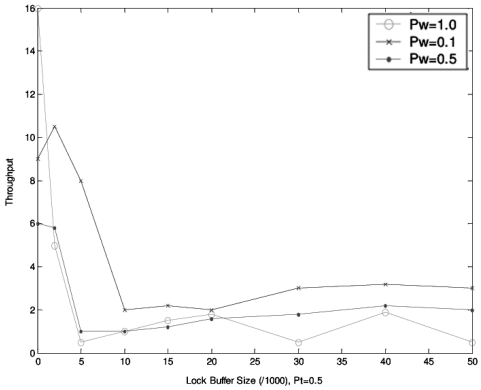


Figure 8: Throughput (Completed Transactions/Sec) Pt = 0.5, $|T| = 1000$

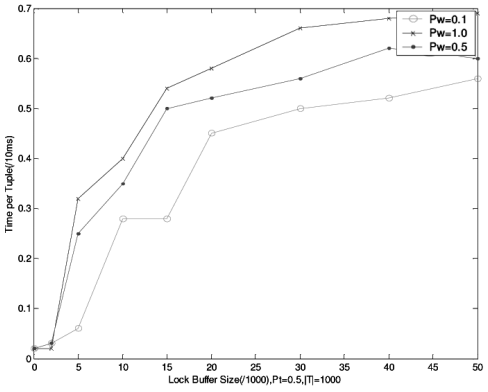


Figure 9: Time per Tuple (/10mns), Pt = 0.5, $|T| = 1000$

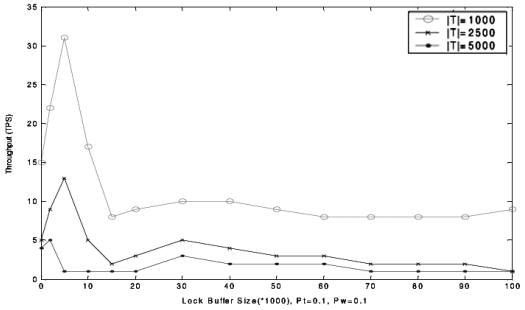


Figure 10: Throughput (Completed Transactions/Sec) Pw = 0.1,

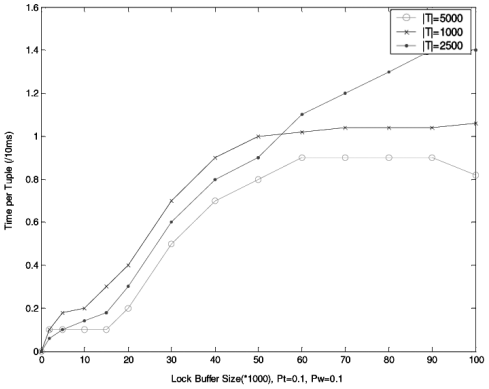


Figure 11: Time per Tuple (/10mns), Pw = 0.1, Pt = 0.1

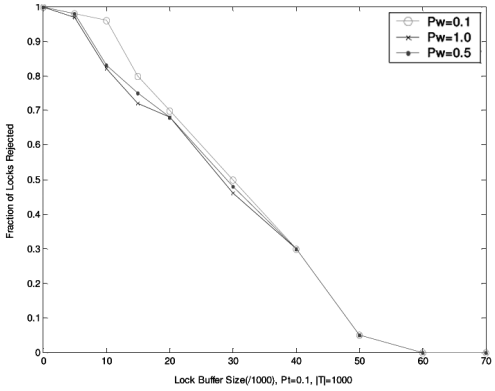


Figure 12: Fraction of Locks Rejected Pt = 0.1, $|T| = 1000$

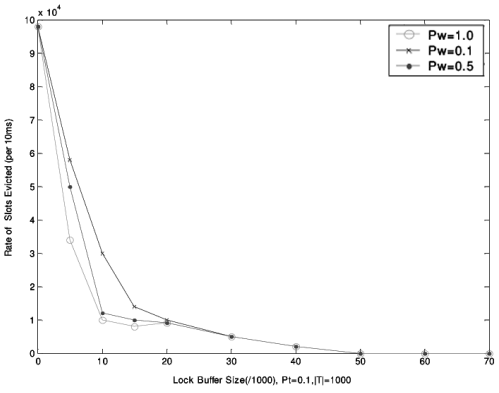


Figure 13: Rate of Slots Evicted (per 10mns), Pt = 0.1, $|T| = 1000$

not affect any of the observations. Thus, the x-axis of each graph is truncated at this point. Also note that for purely optimism (i.e. lock buffer size of 0), we do not report either lock rejection or slot eviction rate. Thus, for these graphs the x-axis starts at 1 and not 0. This again is an artifact of our simulation, which for efficiency reasons bypasses the lock buffer subsystem for purely optimistic accesses. Since the lock buffer subsystem maintains the statistics for rejection rates, it is currently not possible to gather these statistics for a purely optimistic system.

6.2.2 Measure of the Lock Buffer Performance

Figures 6, 8 and 10 each shows a spike in performance between lock buffer sizes of 1000 and 10000 (i.e. 1% to 10% of the database size). These spikes show a throughput increase of up to 4 to 5 times for low conflict rates. As expected spikes become less pronounced with increasing conflict rates (i.e. increasing probability of a write). Thus, we conclude lock buffers enhance performance as long as conflict rates are low. On the other hand Figures 7, 9, and 11, shows that as the lock buffer size increases the time per tuple access increases. We can see that as the system becomes more pessimistic the response time increases. The initial improvement in performance is thus obtained at the cost of increased response time.

For our choice of parameters, pure optimism wins over pure pessimism. However, other experiments we have conducted indicate that for larger transaction sizes and larger write probabilities this is not true. Moreover, under those circumstances lock buffers usually end up performing worse than either pure pessimism or optimism. A hint of this behaviour can be seen in Figures 6 and 11 where dips can be seen in the throughput figures after optimal lock buffer size is reached. This can be explained as follows: our system forces each transaction to first be subjected to the overheads of locking and then to the overheads of being optimistically rolled back. If the lock buffer is not so large more transactions get blocked waiting for locks and are then optimistically rejected. The time spent blocked increases the collision cross section of the transaction subjecting it to more optimistic invalidations. However, after a point when the lock buffer is large enough, the number of optimistic invalidations drops, hence restoring throughput. This shows that the lock buffer must be carefully tuned to achieve optimal performance. However, the dip is only seen when the lock buffer size is roughly twice the optimal size, giving a system designer enough flexibility in designing the system.

6.2.3 Measure of Throughput using varied Transaction Sizes

Figure 11 shows our results for varying transaction sizes with both ProbWrite and ProbReqWriteset to 0:1. The results mimic the first set of graphs with ProbWriteset to 0.1 and ProbReqWriteset to 0.1. As expected, the performance is best for small transaction sizes. Note that smaller transactions have smaller collision cross sections, where the number of locks rejected per slot buffer size drops more rapidly for smaller transaction sizes. Also note that for transactions of size greater than 5000, the system becomes pessimistic only when the lock buffer size is equal to database size. For transaction sizes of 1000, 2500 and 5000 the system reaches optimal throughput when the lock buffer size is 5000 (i.e. 5% of the database size).

6.2.4 Observed Anomalous Behaviour of the System

The only anomalous behaviour seems to be present in Figure 11. In that figure, the graph for transaction size 2500 shows extremely high access time per tuple. This is the only graph that shows such a significant increase. This behaviour was spotted only after the access times were scaled down by the transaction sizes. This behaviour is yet to be accounted for. However, our other experiments indicated that this behaviour is consistent even when we increase the probability of a read-write transaction and probability of a write request.

7. CONCLUSION AND FUTURE RESEARCH DIRECTIONS

In this paper, we have described the analysis of an enhanced optimistic locking model for concurrency control in a distributed database system. In particular, this technique enhances the basic optimistic concurrency model by using locks for high conflict data items. It would be particularly useful in systems that have a few data items that are prone to high rates of conflict. As the empirical results demonstrate, the performance of optimistic concurrency control techniques can be significantly improved by using a relatively small lock buffer. Thus the hybrid technique achieves our goals of improving the performance of optimistic concurrency control and in many cases it also performs better than 2PL.

In future, we will investigate some of the other characteristics of distributed database systems, other than concurrency, like replication and fault-tolerance. We will also explore other types of distributed database systems like Real-Time and Object-Oriented database systems. Finally, we will consider lock buffer tuning as we assumed a fixed size lock buffer in this paper.

REFERENCES

BELL, D. and GRIMSON, J. (1992): Distributed database systems. Addison-Wesley, Workingham.

BERNSTEIN, P., HADZILACOS, V. and GOODMAN, N. (1987): Concurrency control and recovery in database systems. Addison-Wesley.

CAREY, M.J. and LINVY, M. (1991): Conflict detection trades for replicated data. *ACM Transactions on database systems*, 16, (4): 703–746.

DATE, C.J. (1991): An introduction to database systems. Addison Wesley. *Reading*, Fifth Edition.

ELMASRI, R. and NAVATHE, S. (2000): Fundamentals of database systems. Benjamin/Cummings, Third Edition.

GRAY, J.N. and REUTE, A. (1992): Transaction processing: concepts and facilities, Morgan-Kaufmann.

HALICI, U. and DOGAC, A. (1991): An optimistic locking technique for concurrency control in distributed databases. *IEEE Transactions on Software Engineering*, 17 (7): 712–724.

IMIELINSKI, T. and BADRINATH, B.R. (1994): Mobile wireless computing: Challenges in data management. *Technical Report DCS-TR*, Department of Computer Science, Rutgers University.

KOHLER, W. and JENG, B. (1985): Performance evaluation of integrated concurrency control and recovery algorithm using a distributed transaction processing testbed. *Technical Report No. CS-85-133*, Department of Electronics and Computer Engineering, University of Massachusetts, Amherst.

NICOLA, M. and JARKE, M. (2000): Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(4): 645–672.

NORVAG, K., SANDSTA, O. and BRATBERGSENGEN, K. (1997): Concurrency control in distributed object-oriented database systems. *Proceeding of the first East-European Symposium on Advances in Databases and Information Systems*, St. – Petersbury, Russia, (1)

OZSU, M.T and VALDURIEZ, P. (1991): Principle of distributed database systems, Prentice-Hall, Englewood Cliffs, NJ.

RASHMI, S. (2002): Network – Aided concurrency control in a distributed database. PhD. Dissertation, Faculty of School of Engineering and Applied Science, University of Virginia.

SHEIKH, F. and WOODSIDE, C.M. (1997): Layered analytic performance modeling of a distributed database system. *Proceedings of International Conference on Distributed Computer Systems*, 482–490.

THOMASIAN, A. (1998): Concurrency control: Methods performance, and analysis. *ACM Computing Surveys*, 30 (1): 70–119.

YU, P. and DIAS, D. (1992): Analysis of hybrid concurrency control schemes for a high data contention environment. *IEEE Transactions on Software Engineering*, 18 (2): 118–129.

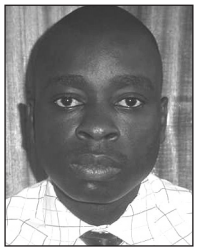
YU, P., WU, K., LIN, K. and SON, S. (1994): On real–time databases: Concurrency control and Scheduling, *Proceedings of the IEEE*, 82(1): 140–157.

BIOGRAPHICAL NOTES

G.A. Aderounmu holds a research degree MSc and PhD in Computer Science from Obafemi Awolowo University, Ile-Ife, Nigeria (1997 and 2001, respectively). He is a member of the Nigerian Society of Engineers (NSE) and is also a registered computer engineer with Council for the Regulation of Engineering Practice in Nigeria (COREN). He is also a Member of the Nigerian Computer Society (NCS) and Computer Professional Registration Council of Nigeria (CPN). He has over 12 years of experience in teaching and research. He is an author of many journal articles in Nigeria and abroad. His special interests include engineering education in Nigeria, curriculum development, and computer communication and network. He is a Visiting Research Fellow to the University of Zululand, Republic of South Africa. He is currently a Senior Lecturer and the Acting Head of the Department of Computer Science and Engineering at the same university.

G.A. Aderounmu

A.A. Akintola obtained his Bachelors and Masters degrees in Computer Engineering and Computer Science from Obafemi Awolowo University, Ile-Ife, in 1994 and 2002, respectively. He is a registered computer engineer with the Council for the Regulation of Engineering Practice in Nigeria (COREN) and also a member of Nigerian Society of Engineers (NSE). He is an author of many journal articles in Nigeria and abroad. His current research interests are in the areas of teletraffic engineering, mobile/nomadic computing, digital computer

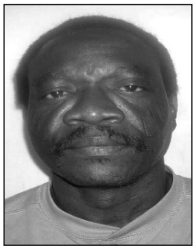


A.A. Akintola

networks, hardware system studies, and computer modeling and simulation of streaming video. He is also into the areas of wireless IP networks, error control, and curriculum development. He has over 7 years of experience in teaching and research. He is currently a PhD student and a lecturer at the Department of Computer Science and Engineering at the same university.

A.U. Osakwe

A.U. Osakwe obtained her Bachelor of Science degree in Computer Engineering from Obafemi Awolowo University, Ile-Ife, in 2003. She is a student member of the Nigerian Society of Engineers (NSE). Her current research interests are in the areas of computer communications and distributed systems. She is also into teletraffic engineering and wireless communications. Miss Osakwe is currently on National Youth Service.



M.O. Adigun

M. O. Adigun holds a research degree PhD in Computer Science from Obafemi Awolowo University, Ile-Ife, Nigeria which he obtained in 1989. Currently, he is a Professor and Head, Department of Computer Science, University of Zululand, Republic of South Africa. He is an author of many journal articles in Nigeria and abroad. His research interests include Software Engineering, Mobile Computing, Modeling and Simulation, and Performance Analysis of Computer Systems.